Broadview®
www.broadview.com.cn

自己动手写Docker

陈显鹭　王炳燊　秦妤嘉◎著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

陈显鹭

阿里云高级研发工程师，对 Docker 有深入研究，是 Docker 多个项目的 Contributor，专注于容器技术的编排与基础环境研究。爱好折腾源代码，热爱开源文化并积极参与社区开源项目的研发。

王炳燊

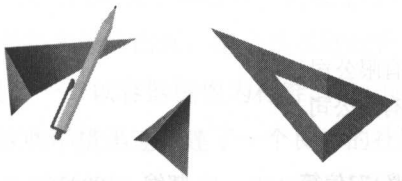
阿里云研发工程师，具有丰富的 Linux 开发经验，对 Docker 有深入研究，多次提交 Docker Patch。目前从事阿里云容器服务网络方案的设计与实现，专注于容器技术的基础环境研究。

秦妤嘉

阿里云高级研发工程师、DevOps 工程师，有丰富的容器化持续集成和持续交付开发实战经验，进行过 Jenkins 源码分析改造和 Jenkins 插件开发。目前从事阿里云容器服务持续集成和持续交付方案的设计和实现。

自己动手写Docker

陈显鹭　王炳燊　秦妤嘉◎著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书在详细分析 Docker 所依赖的技术栈的基础上，一步一步地通过代码实例，让读者可以自己循序渐进地用 Go 语言构建出一个容器的引擎。不同于其他 Docker 原理介绍或代码剖析的书籍，本书旨在提供给读者一条动手路线，一步一步地实现 Docker 的隔离性，构建 Docker 的镜像、容器的生命周期及 Docker 的网络等。本书涉及的代码都托管在 GitHub 上，读者可以对照书中的步骤从代码层面学习构建流程，从而精通整个容器技术栈。本书也对目前业界容器技术的方向和实现做了简单介绍，以加深读者对容器生态的认识和理解。

本书适合对容器技术已经使用过或有一些了解，希望更深层次掌握容器技术原理和最佳实践的读者。

图书在版编目（CIP）数据

自己动手写Docker / 陈显鹭，王炳燊，秦妤嘉著. —北京：电子工业出版社，2017.7

ISBN 978-7-121-31786-6

Ⅰ. ①自… Ⅱ. ①陈… ②王… ③秦… Ⅲ. ①Linux操作系统－程序设计 Ⅳ. ①TP316.85

中国版本图书馆CIP数据核字（2017）第124163号

策划编辑：张春雨
责任编辑：徐津平
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　印张：13.25　字数：269千字
版　　次：2017年7月第1版
印　　次：2017年7月第1次印刷
定　　价：65.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

序

我是阿里云容器服务团队的架构师易立，很荣幸为这本书作序。

当显鹭等几位同学跟我谈起他们想写一本介绍如何从头打造一个 Docker 引擎的书时，我有些担心这样的内容是不是太小众，毕竟绝大多数读者都是 Docker 的使用者而非开发者。然而读完样章，看到这三位同学笔下翔实的内容，文中透出的热情和自信打消了我的顾虑。

Docker 是技术圈中的当红小鲜肉。自从 2013 年横空出世以来，迅速在开发者社区流行开来。在 2016 年 9 月，Docker 镜像在 Docker Hub 的总下载量就已经超过了 60 亿次，并且以每 6 周 10 亿次的速度迅速增长。

大家都知道 Docker 技术脱胎于 Linux Container（LXC）技术，在 LXC 的发展过程中，IBM、Google、Red Hat、Canonical 等技术巨擘做出了众多的贡献。然而，Docker 到底有什么魔力，能够在这么短的时间之内就风靡了整个技术圈呢？

Docker 公司的创始人兼 CTO——Solomon Hykes，有机地把一系列技术 Cgroups、Namespace 和 UnionFS 整合起来，极大地降低了容器技术的复杂度，提升了开发者的用户体验。他敏锐地预测到，一旦标准化容器技术最终出现，整个技术行业将会受到深远的影响。Docker 公司开源了 Docker Engine，定义了一个以容器镜像为标准的应用打包格式，并且建立 Docker Hub 服务进行镜像分发和协作。这些举措迅速创建了一个良好的社区和合作伙伴生态圈，包含 AWS、Google、Microsoft、IBM 和国内的众多公司。在短短几年的时间内，Docker 几乎成为了容器技术的代名词。

“得标准者得天下”，容器底层标准化之争风云再起。2014 年年底，CoreOS 推出 rkt 容器引擎，试图挑战 Docker 另立标准。Docker 在 2015 年 6 月宣布成立 OCI（Open Container Initiative）组织作为 Linux 基金会的协作项目，并将其容器标准和 runtime 参考实现（runC）贡献出来，旨在围绕容器格式和运行时制定一个开放的工业化标准。这一举措化解了社区在容器标准上的第一次分歧。

随着容器技术的快速发展，技术生态逐渐从围绕单机环境构建和运行容器化应用，发展为支持大规模容器编排技术。云平台成为了分布式网络操作系统，而容器成为了“进程”执行单元，可以动态地运行在不同宿主机环境中。其中，Kubernetes、Mesos、Docker 诸强争霸，各有所长。2016 年 6 月，Docker 宣布开始在 Docker Engine 中内置 Swarm mode，这极大地简化了容器编排的复杂性，但也遭到了社区的强烈反对。Google 发起 CRI（Container Runtime Interface，容器运行时接口）项目，通过 shim 的抽象层使得调度框架支持不同的容器引擎实现。Mesos 推出了 Unified Containerizer，以支持 Docker、Appc、OCI 等不同的镜像格式，而无须再依赖 Docker Engine。

面对这些挑战，2016 年 12 月 14 日，Docker 公司宣布将 Docker Engine 的核心组件 Containerd 捐赠到一个新的开源社区，任其独立发展和运营，目标是提供一个标准化的容器 runtime，其注重简单、健壮性和可移植性。由于 Containerd 只包含最基本的容器管理能力，因此上层框架可以有更大的灵活性来提供容器的调度和编排能力。阿里云、AWS、Google、IBM 和 Microsoft 作为 Containerd 的初始成员，为项目贡献力量。

在技术爆发的年代，新技术层出不穷，而快餐式的阅读和了解无法帮助我们梳理和把握发展的脉络。对一些核心技术既要知其然也要知其所以然，这样才能举一反三，对技术趋势建立起自己的理解和判断。了解容器基础知识，可以深入理解容器在进程管理、资源管理、安全隔离等方面与传统方式的不同，也有助于了解容器在网络、存储、安全等方面的特殊性。

最好的学习方式莫过于自己亲手实践。计算机界的泰斗 Andrew Tanenbaum 教授为教学而构建了 Minix，而这也启发了 Linus Torvalds 大神创造了 Linux。我们期待同学们能够从本书循序渐进的讲解中学习容器相关的技术细节，深入理解 Docker 的底层技术实现，围绕容器技术实现创造性的扩展和应用。

易立

2017 年 1 月

前言

为什么要写这本书

Docker 技术可谓是近年最火热的技术之一，铺天盖地的技术论坛和各种讲座，大家都在分享关于如何容器化及如何使用 Docker 优化自己运维和开发流程的经验。随着 Docker 技术的逐渐普及，使用 Docker 已经不再是一个难题。现在更加重要的是生产环境容器化的最佳实践，另外就是容器的编排框架之争。但是，对于技术人员来说，除去 Docker 外表的繁华，什么是容器，容器到底是怎么创建的，容器底层的技术探秘也是非常重要的。

我在 2014 年开始接触 Docker，经历了从最初的新奇——感叹竟然还有 Docker 这样的好工具，到逐渐熟悉 Docker 的各种功能，尝试在生产环境中使用 Docker 技术的过程。但是，每每被人问到："Docker 技术到底是怎么实现的呢？"我只能粗粗浅浅地说："Docker 是使用 Linux Kernel 的 Namespace 和 Cgroups 实现的一种容器技术。"那么，什么是 Namespace，什么是 Cgroups，Docker 是怎么使用它们的，容器到底是怎么一步步被创建出来的？问到这些，我就会支支吾吾地不知所以。由此可见，了解容器技术的底层技术，然后明白它们是如何工作的，尤为重要，这些才是整个容器技术的基石，掌握了这些基石才能更加容易地向上攀登。

单单讲解底层的技术实现细节和源码解读是很枯燥的一件事，一般来说很难有耐心去一点点细读然后揣摩其中的奥妙，这样囫囵吞枣地过一遍技术细节，作用不大。因此，我便萌生了写一本《自己动手写 Docker》这样的书的想法。本书不去刻意讲解容器技术的细节，用到什么讲解什么，点到为止，更加细节的内容留给读者自己探索。通过阅读本书，可以一步步地了解容器技术的实现细节，更可以一步步地用自己的代码去实现它。本书最大的乐趣莫过于用自己最新了解到的知识去动手实现自己的容器。由此可以进一步打开你进入容器技术社区的大门。

本书的内容

本书的目的是去引导读者通过学习容器技术的实现细节，一步步去构建一个简单的容器。从这个过程中，了解整个容器技术领域和实现细节。本书注重原理的讲解与实践，每一部分都会有详细的代码解析，力争用最少、最精简的代码，帮助读者构建自己的容器。

本书的内容主要分为“容器与开发语言”“基础技术”“构造容器”“构造镜像”“构造容器进阶”“容器网络”“高级实践”这 7 章。

- 容器与开发语言：主要介绍 Docker 的基本功能和特点，并且对后面即将使用的 Go 语言做一个简单的介绍。
- 基础技术：主要介绍实现容器的底层技术，如 Namespace、Cgroups、Union File System。每一节都会有文字性介绍，并且附有一个简短的小例子程序，介绍在容器上是如何使用这项技术的，方便读者清晰地理解各个技术点在容器上的作用。
- 构造容器：使用前面两章介绍的基础技术，构造一个最简单的容器环境，会将整体实现细节及代码解析一点点展现，直接使用前面介绍的基础技术，从而更加有实战感。
- 构造镜像：使用 2.3 节介绍的分层文件系统技术，构建一个简单的容器镜像，体现容器镜像的分层思想。
- 构造容器进阶：更加贴近真实的容器实现，在原来的基础上，增加更丰富的功能。通过这一章的学习，读者可以更好地了解各种技术是如何整合在一起去实现容器整体功能的。
- 容器网络：除实现一个容器环境之外，这一章还会讲解如何使自己的容器和宿主机通信，以及如何让不同的容器之间进行通信，更加贴近真实环境。
- 高级实践：使用自己编写的容器，运行一些通用程序，验证容器的可用性。此外，本章还介绍了目前 Docker 使用的容器运行引擎，以及目前容器运行态引擎的概况。

适用读者

- 希望更加深入地了解容器技术的读者。
- 已经使用过 Docker，希望探查细节的读者。
- Go 语言程序员，了解如何使用 Go 语言来编写自己的容器。
- 容器技术爱好者。

如何阅读

由于本书的定位是自己动手写 Docker，侧重于实战，因此仅仅纸面阅读是无法体会所有要点的。所有的源码均托管在 http://github.com/xianlubird/mydocker，您可以在这里下载到本书的所有源码。每一章节都会有一个对应的 tag，建议您在阅读书中讲解内容的同时，也将代码下载下来，在本机尝试运行，了解整个代码的运行流程。我们非常欢迎向本项目提交 Pull Request，在阅读思考中不断交流学习。

关于勘误

由于时间和水平都比较有限，因此本书难免会存在一些纰漏和错误。如果读者发现了问题，请及时与我们联系，我们也会在后面的版本中加以改正，邮箱是 xianlubird@gmail.com。非常希望与大家共同学习容器技术。

致谢

最后，向本书编写过程中给予我们巨大帮助的人们表示诚挚的感谢。感谢女友的支持，没有她包揽所有家务，我不会有大量空余时间编写此书。感谢同事姜继忠和谢瑶瑶对于本书 containerd 和 Kubernetes 相关章节内容的支持。感谢阿里云容器服务团队在本书编写期间给予的理解与包容。最后，感谢张春雨编辑，是他的帮助与支持才使得本书由一个想法变成了实体，展现给各位读者。

陈显鹭

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/31786*

目录

第1章 容器与开发语言

1.1 Docker

最近一段时间，云计算领域最火的莫过于“容器”一词。提到容器，就不得不提 Docker，可以说 Docker 已经成为了容器的代名词。那么，什么是 Docker？Docker 又能做什么呢？本章我们就来简单介绍一下 Docker。

1.1.1 简介

Docker 是一个开源工具，它可以将你的应用打包成一个标准格式的镜像，并且以容器的方式运行。Docker 容器将一系列软件包装在一个完整的文件系统中，这个文件系统包含应用程序运行所需要的一切：代码、运行时工具、系统工具、系统依赖，几乎有任何可以安装在服务器上的东西。这些策略保证了容器内应用程序运行环境的稳定性，不会被容器外的系统环境所影响。

图 1.1 是一个容器镜像的结构图。从中可以看到，镜像可以把系统级依赖都打包成一个文件，所有的容器会共享一个 Kernel，因此在同一个 Kernel 下可以运行各种 Linux 发行版的容器。

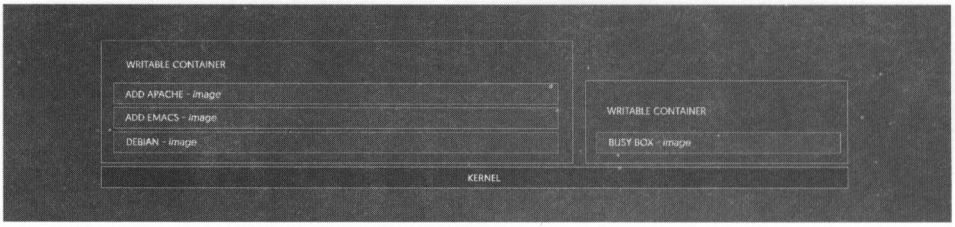


图 1.1

Docker 容器具有以下 3 个特点。

- 轻量级：在同一台宿主机上的容器共享系统 Kernel，这使得它们可以迅速启动而且占用内存极少。镜像是以分层文件系统构造的，这可以让它们共享相同的文件，使得磁盘使用率和镜像下载速度得到提高。
- 开放：Docker 容器基于开放标准，这使得 Docker 容器可以运行在主流 Linux 发行版和 Windows 操作系统上。
- 安全：容器将各个应用程序隔离开来，这给所有的应用程序提供了一层额外的安全防护。

1.1.2　容器和虚拟机比较

容器和虚拟机同样有着资源隔离和分配的优点，但是由于其架构的不同，容器比虚拟机更加便携和高效。

虚拟机包含用户的程序，必要的函数库和整个客户操作系统，所有的这些差不多需要占用好几个 GB 的空间。图 1.2 为虚拟机的架构图。

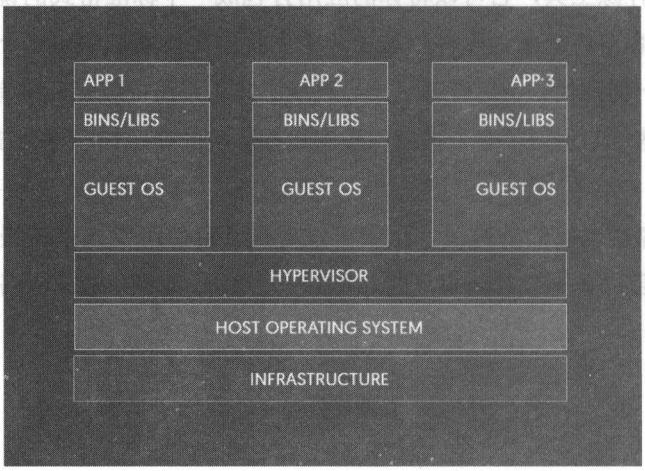


图 1.2

容器包含用户的程序和所有的依赖，但是容器之间是共享 Kernel 的。各个容器在宿主机上互相隔离，并且在用户态下运行。Docker 容器不和任何基础设施绑定，它可以运行在任何电脑、IDC 和云上。图 1.3 为 Docker 容器的架构图。

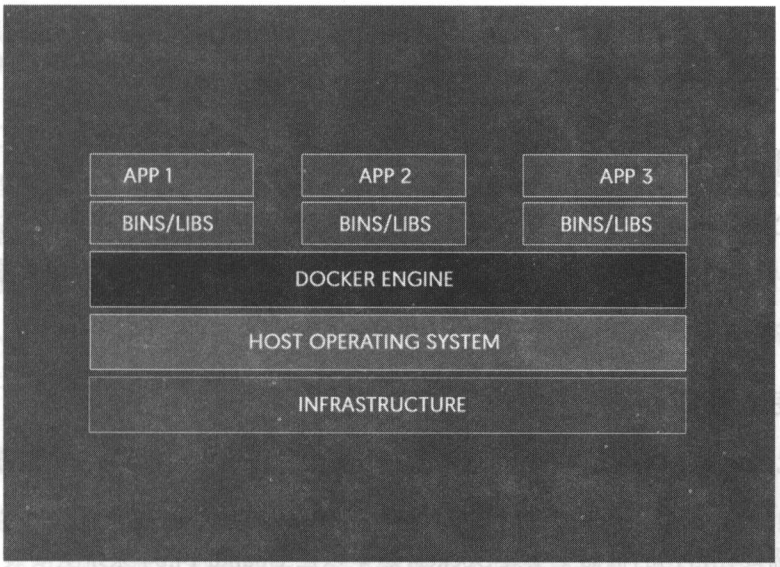


图 1.3

1.1.3　容器加速开发效率

Docker 容器可以帮助开发者跳过设置冗杂的开发环境，专注于开发软件的新功能，具体有如下 3 项。

- 加速开发：再也不用等待数小时设置开发环境，可以很方便地使生产环境的代码在本地跑起来。
- 赋能创造力：Docker 容器的隔离特性可以让开发者摆脱限制。开发者可以为自己的应用选择最好的语言和工具，再也不用担心产生内部工具的冲突。
- 消除环境不一致：将应用程序的配置和所有依赖打包成一个镜像在容器中，可以保证应用在任何环境中都可以按照预期来运行，再也不用担心不得不在不同环境中安装相同软件和配置的问题。

1.1.4　利用容器合作开发

Docker 镜像可以存储到 Docker Hub 中，团队成员可以通过 Docker Store、Docker Hub 管理分享镜像。所有的变化和历史都可以在整个组织间查看。

而且，你可以很简单地分享你的容器，不需要担心环境依赖产生的不一致问题，其他团队也可以很简单地引用你的容器，而不需要去关心它是如何工作的。

1.1.5　利用容器快速扩容

Docker 允许动态地改变应用程序，可以通过扩容快速提高应用程序的能力并及时修复缺陷。Docker 容器可以秒级启动和停止，因此，它可以在需要的时候快速扩容出大量的应用程序，扛住并发的压力。

1.1.6　安装使用 Docker

Docker 就是一个这样的工具。它可以帮助开发者很方便地去构建、部署、运行自己的程序，还可以让你非常迅速地测试你的项目并将其部署到生产环境中。

首先，你需要在自己的机器上安装 Docker，这里以在 Ubuntu 14.04 系统上安装 Docker 为例。

```
curl -sSL https://get.docker.com | sh
```

运行以上命令，一段美妙的小脚本就这样被安装到了你的机器上，它完成了安装 Docker 需要的所有内容。下面，就开始使用它吧。以安装一个 WordPress 为例，看看 Docker 是如何快速安装一个 WordPress 的。以前安装 WordPress，可能需要去了解 PHP、MySQL，还有服务器的系统，最后才去安装 WordPress，非常麻烦。但是，如果换一种方式，使用 Docker 来安装呢？

```
docker run -d -p 80:80 --name wordpress wordpress
```

运行以上命令，Docker 会自动从 Docker Hub 中拉取 WordPress 镜像，这个镜像是已经被 build 好的，包含了 PHP、MySQL 和 WordPress。你所做的工作就是等待 Docker 帮你把这个服务启动起来以后，在浏览器上访问你的服务器 IP，就可以看到 WordPress 的安装页面，然后一步步点击页面按钮完成安装即可。对于 MySQL 密码，可以使用如下命令获取。

```
echo $(docker logs wordpress | grep password)
```

上面这条命令就可以获得 MySQL 密码，将其填写到网页中，就得到了一个可以运行的 WordPress，然后开始愉快地使用它吧。

是不是感受到了 Docker 的威力？其实这只是 Docker 强大功能的冰山一角。快速部署是 Docker 其中的一个特性。你不需要登录到服务器，将运行环境一个一个地安装好，最后再部

署自己的代码。Docker 像集装箱一样，帮助你打包好了一切，你只需要开箱使用即可。就像刚才的例子，还可以非常简单地再次运行刚才的命令，只需要换一下映射的端口，就可以再启动一个 WordPress，这是安装原生应用所不敢想象的。

1.2　Go

Go 语言又称 Golang，是 Google 开发的一种静态强类型、编译型、并发型并具有垃圾回收功能的编程语言。Go 语言在 2009 年第一次被披露，并在 2012 年发布了 1.0 版本，可以说是一门非常年轻的语言。Go 语言的创造者可谓众星云集，包括 UNIX 操作系统和 B 语言（C 语言的前身）的创造者、UTF-8 编码的发明者 Ken Thompson，UNIX 项目的参与者、UTF-8 编码的联合创始人和 Limbo 编程语言（Go 语言的前身）的创造者 Rob Pike，以及著名的 JavaScript 引擎 V8 的创造者 Robert Griesemer。

1.2.1　描述

Go 语言的语法虽然接近 C 语言，但还是有一些不同，比如两者对于变量的声明是不同的，且 Go 语言中的 for 循环和 if 判断语句不需要用小括号括起来。Go 语言的并行模型是以东尼·霍尔的通信顺序进程（CSP）为基础的，并采取了类似模型的其他语言（包括 Occam 和 Limbo），但它也具有 Pi 运算的特征，比如通道传输。

与 C++ 相比，Go 语言并不包括如异常处理、继承、泛型、断言、虚函数等功能，但增加了 slice 型、并发、管道、垃圾回收、接口（Interface）等特性的语言级支持。当然，Google 对于泛型的态度还是很开放的，但在该语言的常见问题列表中，对于断言的存在，则持负面态度，同时也在为自己不提供类型继承辩护。不同于 Java，Go 语言内嵌了关联数组（也称为哈希表（Hash）或字典（Dictionary）），就像字符串类型一样。

可以在 Go 语言官网首页看到一个 Go 语言的 Hello World 示例，代码如下。

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

目前使用 Go 开发的项目非常多，其中就有国人开发的 beego（用来开发 Go 应用程序的开源框架），另外一个就是大名鼎鼎的 Docker。因此，本书会以 Go 语言代码为示例开发我们自己的 Docker 应用。

1.2.2　安装 Go

本书的开发测试环境如下。

- Ubuntu 14.04。
- 内核版本 3.13.0-83-generic。
- Go 版本 1.7.1。

可以在 Go 语言官网 https://golang.org/dl/ 根据操作系统下载对应的安装包。这里以 Linux 为例进行安装，首先下载安装包 go1.7.1.linux-amd64.tar.gz，然后执行 tar -C /usr/local -xzf go1.7.1.linux-amd64.tar.gz，将安装包解压到 /usr/local 目录下。编辑 $HOME/.profile 或 $HOME/.bashrc，将 export PATH=$PATH:/usr/local/go/bin 命令添加到文件中，然后执行 source $HOME/.bashrc，使修改生效。这时就可以在系统中使用 Go 命令了，执行 go version 来看一下，命令如下。

```
root@vagrant-ubuntu-trusty-64:~# go version
go version go1.7.1 linux/amd64
```

1.2.3　配置 GOPATH

GOPATH 是真正存放代码的路径，Go 寻找依赖包时会根据 $GOPATH 来寻找，GOPATH 目录约定有如下 3 个子目录。

- src 存放源代码。
- pkg 存放编译后生成的文件。
- bin 存放编译后的可执行文件。

这里以 /go 为 GOPATH 路径，编辑 ~/.bashrc 文件，将命令 export GOPATH=/go 添加到文件中，然后执行 source ~/.bashrc，之后再执行 go env 看一下效果，结果如下。

```
root@vagrant-ubuntu-trusty-64:/go/src# go env
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/go"
```

```
GORACE=""
GOROOT="/usr/local/go"
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-
build735379415=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"
```

可以看到，$GOPATH 已经被指定了。

本书中的代码都会基于以上配置，代码路径为 $GOPATH/src/github.com/xianlubird/mydocker，项目名称为 mydocker，后面会基于这个路径进行开发。

1.3　小结

本章主要介绍了什么是 Docker，以及使用容器技术带来的优势。之后介绍了如何安装和使用 Docker 做一个简单的 demo。同时简单讲解了 Go 语言的不同，以及基本的开发环境配置与使用。下面，将以此为基础进行自己的容器开发。

第2章 基础技术

2.1 Linux Namespace 介绍

我们经常听到，Docker 是一个使用了 Linux Namespace 和 Cgroups 的虚拟化工具。但是，什么是 Linux Namespace，它在 Docker 内是怎么被使用的？说到这里，很多人就会迷茫。下面就先来介绍一下 Linux Namespace 及它们是如何在容器中使用的。

2.1.1 概念

Linux Namespace 是 Kernel 的一个功能，它可以隔离一系列的系统资源，比如 PID（Process ID）、User ID、Network 等。一般看到这里，很多人会想到一个命令 chroot，就像 chroot 允许把当前目录变成根目录一样（被隔离开来的），Namespace 也可以在一些资源上，将进程隔离起来，这些资源包括进程树、网络接口、挂载点等。

比如，一家公司向外界出售自己的计算资源。公司有一台性能还不错的服务器，每个用户买到一个 tomcat 实例用来运行它们自己的应用。有些调皮的客户可能不小心进入了别人的 tomcat 实例，修改或关闭了其中的某些资源，这样就会导致各个客户之间互相干扰。也许你会说，我们可以限制不同用户的权限，让用户只能访问自己名下的 tomcat 实例，但是，有些操作可能需要系统级别的权限，比如 root 权限。我们不可能给每个用户都授予 root 权限，也不可能给每个用户都提供一台全新的物理主机让他们互相隔离。因此，Linux Namespace 在这里就派上了用场。使用 Namespace，就可以做到 UID 级别的隔离，也就是说，可以以 UID 为 n 的用户，

虚拟化出来一个 Namespace，在这个 Namespace 里面，用户是具有 root 权限的。但是，在真实的物理机器上，他还是那个以 UID 为 n 的用户，这样就解决了用户之间隔离的问题。当然这只是 Namespace 其中的一个简单功能。

除了 User Namespace，PID 也是可以被虚拟的。命名空间建立系统的不同视图，从用户的角度来看，每一个命名空间应该像一台单独的 Linux 计算机一样，有自己的 init 进程（PID 为 1），其他进程的 PID 依次递增，A 和 B 空间都有 PID 为 1 的 init 进程，子命名空间的进程映射到父命名空间的进程上，父命名空间可以知道每一个子命名空间的运行状态，而子命名空间与子命名空间之间是隔离的。从图 2.1 所示的 PID 映射关系图中可以看到，进程 3 在父命名空间中的 PID 为 3，但是在子命名空间内，它的 PID 就是 1。也就是说用户从子命名空间 A 内看进程 3 就像 init 进程一样，以为这个进程是自己的初始化进程，但是从整个 host 来看，它其实只是 3 号进程虚拟化出来的一个空间而已。

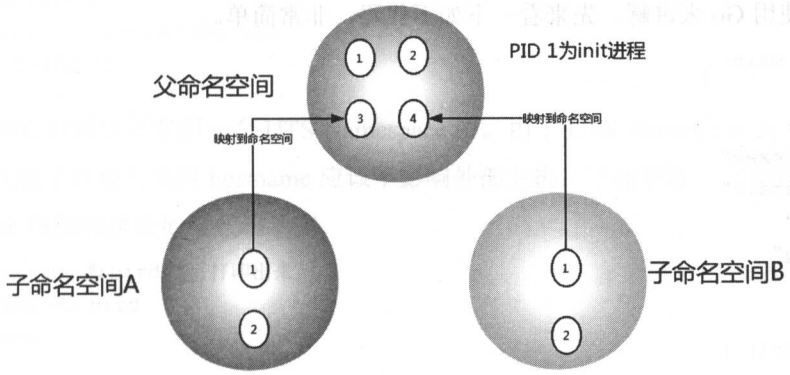


图 2.1

当前 Linux 一共实现了 6 种不同类型的 Namespace。

Namespace 类型	系统调用参数	内核版本
Mount Namespace	CLONE_NEWNS	2.4.19
UTS Namespace	CLONE_NEWUTS	2.6.19
IPC Namespace	CLONE_NEWIPC	2.6.19
PID Namespace	CLONE_NEWPID	2.6.24
Network Namespace	CLONE_NEWNET	2.6.29
User Namespace	CLONE_NEWUSER	3.8

Namespace 的 API 主要使用如下 3 个系统调用。

- clone() 创建新进程。根据系统调用参数来判断哪些类型的 Namespace 被创建，而且它们的子进程也会被包含到这些 Namespace 中。
- unshare() 将进程移出某个 Namespace。
- setns() 将进程加入到 Namespace 中。

2.1.2 UTS Namespace

UTS Namespace 主要用来隔离 nodename 和 domainname 两个系统标识。在 UTS Namespace 里面，每个 Namespace 允许有自己的 hostname。

下面将使用 Go 来做一个 UTS Namespace 的例子。其实对于 Namespace 这种系统调用，使用 C 语言来描述是最好的，但是本书的目的是去实现 Docker，由于 Docker 就是使用 Go 开发的，所以就整体使用 Go 来讲解。先来看一下如下代码，非常简单。

```
package main

import (
    "os/exec"
    "syscall"
    "os"
    "log"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

解释一下代码，exec.Command（"sh"）用来指定被 fork 出来的新进程内的初始命令，默认使用 sh 来执行。下面就是设置系统调用参数，像 2.1.1 小节中讲到的一样，使用 CLONE_

NEWUTS 这个标识符去创建一个 UTS Namespace。Go 帮我们封装了对 clone() 函数的调用，这段代码执行后就会进入到一个 sh 运行环境中。

在 Ubuntu 14.04 上运行这个程序，Kernel 版本为 3.13.0-65-generic，Go 版本为 1.7.3，执行 go run main.go 命令，在这个交互式环境里，使用 pstree -pl 查看一下系统中进程之间的关系，如下。

```
|-sshd(19820)---bash(19839)---go(19901)-+-main(19912)-+-sh(19915)---
    pstree(19916)
```

然后，输出一下当前的 PID，代码如下。

```
# echo $$
19915
```

验证一下父进程和子进程是否不在同一个 UTS Namespace 中，验证代码如下。

```
# readlink /proc/19912/ns/uts
uts:[4026531838]
# readlink /proc/19915/ns/uts
uts:[4026532193]
```

可以看到它们确实不在同一个 UTS Namespace 中。由于 UTS Namespace 对 hostname 做了隔离，所以在这个环境内修改 hostname 应该不影响外部主机，下面来做一下实验。

在这个 sh 环境内执行如下代码示例。

```
# 修改 hostname 为 bird 然后打印出来
# hostname -b bird
# hostname
bird
```

另外启动一个 shell，在宿主机上运行 hostname，看一下效果。

```
root@iZ254rt8xf1Z:~# hostname
iZ254rt8xf1Z
```

可以看到，外部的 hostname 并没有被内部的修改所影响，由此可了解 UTS Namespace 的作用。

2.1.3　IPC Namespace

IPC Namespace 用来隔离 System V IPC 和 POSIX message queues。每一个 IPC Namespace 都有自己的 System V IPC 和 POSIX message queue。

在上一版本的基础上稍微改动了一下代码。

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

可以看到，仅仅增加 syscall.CLONE_NEWIPC 代表我们希望创建 IPC Namespace。下面，需要打开两个 shell 来演示隔离的效果。

首先在宿主机上打开一个 shell。

```
# 查看现有的 ipc Message Queues
root@iZ254rt8xf1Z:~# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

# 下面创建一个 message queue
root@iZ254rt8xf1Z:~# ipcmk -Q
Message queue id: 0
# 然后再查看一下
root@iZ254rt8xf1Z:~# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x5e8f3f1e 0          root       644        0            0
```

这里，能够发现可以看到一个 queue 了。下面，使用另外一个 shell 去运行程序。

```
root@iZ254rt8xf1Z:~/gocode/src/book# go run main.go
# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

通过以上实验，可以发现，在新创建的 Namespace 里，看不到宿主机上已经创建的 message queue，说明 IPC Namespace 创建成功，IPC 已经被隔离。

2.1.4　PID Namespace

PID Namespace 是用来隔离进程 ID 的。同样一个进程在不同的 PID Namespace 里可以拥有不同的 PID。这样就可以理解，在 docker container 里面，使用 ps -ef 经常会发现，在容器内，前台运行的那个进程 PID 是 1，但是在容器外，使用 ps -ef 会发现同样的进程却有不同的 PID，这就是 PID Namespace 做的事情。

在 2.1.3 小节中代码的基础上，再修改一下代码，添加一个 syscall.CLONE_NEWPID，代表为 fork 出来的子进程创建自己的 PID Namespace。

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC | syscall.CLONE_
        NEWPID,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

我们需要打开两个 shell。首先在宿主机上看一下进程树，找一下进程的真实 PID。

```
root@iZ254rt8xf1Z:~# pstree -pl
 |-sshd(894)-+-sshd(9455)---bash(9475)---bash(19619)
    |           |-sshd(19715)---bash(19734)
    |           |-sshd(19853)---bash(19872)---go(20179)-+-main(20190)-+-sh(20193)
    |           |                                       |             |-{main}(20191)
    |           |                                       |             `-{main}(20192)
    |           |                                       |-{go}(20180)
    |           |                                       |-{go}(20181)
    |           |                                       |-{go}(20182)
    |           |                                       `-{go}(20186)
    |           `-sshd(20124)---bash(20144)---pstree(20196)
```

可以看到，go main 函数运行的 PID 为 20190。下面，打开另外一个 shell 运行一下如下代码。

```
root@iZ254rt8xf1Z:~/gocode/src/book# go run main.go
# echo $$
1
```

可以看到，该操作打印了当前 Namespace 的 PID，其值为 1。也就是说，这个 20190 的 PID 被映射到 Namespace 里后 PID 为 1。这里还不能使用 ps 来查看，因为 ps 和 top 等命令会使用 /proc 内容，具体内容在下面的 Mount Namespace 部分会进行讲解。

2.1.5 Mount Namespace

Mount Namespace 用来隔离各个进程看到的挂载点视图。在不同 Namespace 的进程中，看到的文件系统层次是不一样的。在 Mount Namespace 中调用 mount() 和 umount() 仅仅只会影响当前 Namespace 内的文件系统，而对全局的文件系统是没有影响的。

看到这里，也许就会想到 chroot()。它也是将某一个子目录变成根节点。但是，Mount Namespace 不仅能实现这个功能，而且能以更加灵活和安全的方式实现。

Mount Namespace 是 Linux 第一个实现的 Namespace 类型，因此，它的系统调用参数是 NEWNS（New Namespace 的缩写）。当时人们貌似没有意识到，以后还会有很多类型的 Namespace 加入 Linux 大家庭。

针对 2.1.4 小节中的代码做了一点改动，增加了 NEWNS 标识，如下。

```
package main

import (
    "log"
    "os"
    "os/exec"
```

```
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC | syscall.CLONE_NEWPID |
        syscall.CLONE_NEWNS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

首先，运行代码，然后查看一下 /proc 的文件内容。proc 是一个文件系统，提供额外的机制，可以通过内核和内核模块将信息发送给进程。

```
# ls /proc
1     14    19872  23   34    43   55   739   865   bus          filesystems kpagecount
pagetypeinfo    sysvipc
10    145   2      24   348   44   57   75    866   cgroups      fs          kpageflags
partitions      timer_list
100   1472  20     25   35    45   58   76    869   cmdline      interrupts  latency_stats
sched_debug     timer_stats
11    1475  20124  26   353   47   59   77    894   consoles     iomem       loadavg
schedstat       tty
1174  15    20129  27   36    48   6    776   9     cpuinfo      ioports     locks
scsi            uptime
1192  154   20144  28   37    49   60   78    937   crypto       ipmi        mdstat
self            version
12    155   20215  29   38    5    607  796   945   devices      irq         meminfo
slabinfo        version_signature
1255  16    20226  3    39    50   61   8     9460  diskstats    kallsyms    misc
softirqs        vmallocinfo
1277  17    20229  30   391   51   62   827   967   dma  kcore   modules     stat
vmstat
1296  18    20231  31   40    52   63   836   99    driver       key-users   mounts
swaps           xen
13    19    21     32   41    53   7    860   acpi  execdomains  keys  mtrr  sys
zoneinfo
1309  19853 22     33   42    54   733  862   buddyinfo fb       kmsg  net   sysrq-trigger
```

因为这里的 /proc 还是宿主机的，所以看到里面会比较乱，下面，将 /proc mount 到我们自己的 Namespace 下面来。

```
# mount -t proc proc /proc
# ls /proc
1        consoles    execdomains  ipmi        kpagecount   misc        sched_debug  swaps
uptime
5        cpuinfo     fb           irq         kpageflags   modules     schedstat    sys
version
acpi     crypto      filesystems  kallsyms    latency_stats  mounts    scsi         sysrq-
trigger  version_signature
buddyinfo  devices   fs           kcore       loadavg      mtrr        self         sysvipc
vmallocinfo
bus      diskstats   interrupts   key-users   locks        net         slabinfo     timer_list
vmstat
cgroups  dma         iomem        keys        mdstat       pagetypeinfo  softirqs   timer_stats
xen
cmdline  driver      ioports      kmsg        meminfo      partitions  stat         tty
zoneinfo
```

可以看到，瞬间少了好多文件。下面就可以使用 ps 来查看系统的进程了。

```
# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 20:15 pts/4    00:00:00 sh
root         6     1  0 20:19 pts/4    00:00:00 ps -ef
```

可以看到，在当前的 Namespace 中，sh 进程是 PID 为 1 的进程。这就说明，当前的 Mount Namespace 中的 mount 和外部空间是隔离的，mount 操作并没有影响到外部。Docker volume 也是利用了这个特性。

2.1.6 User Namespace

User Namespace 主要是隔离用户的用户组 ID。也就是说，一个进程的 User ID 和 Group ID 在 User Namespace 内外可以是不同的。比较常用的是，在宿主机上以一个非 root 用户运行创建一个 User Namespace，然后在 User Namespace 里面却映射成 root 用户。这意味着，这个进程在 User Namespace 里面有 root 权限，但是在 User Namespace 外面却没有 root 的权限。从 Linux Kernel 3.8 开始，非 root 进程也可以创建 User Namespace，并且此用户在 Namespace 里面可以被映射成 root，且在 Namespace 内有 root 权限。

下面，继续以一个例子来描述，代码如下。

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
      Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC | syscall.CLONE_NEWPID |
        syscall.CLONE_NEWNS |
            syscall.CLONE_NEWUSER,
    }
    cmd.SysProcAttr.Credential = &syscall.Credential{Uid: uint32(1), Gid:
    uint32(1)}
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
    os.Exit(-1)
}
```

本例在原来的基础上增加了 syscall.CLONE_NEWUSER。首先，以 root 来运行这个程序，运行前在宿主机上看一下当前的用户和用户组，显示如下。

```
root@iZ254rt8xf1Z:~/gocode/src/book# id
uid=0(root) gid=0(root) groups=0(root)
```

可以看到我们是 root 用户，接下来运行一下程序。

```
root@iZ254rt8xf1Z:~/gocode/src/book# go run main.go
$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

可以看到，它们的 UID 是不同的，因此说明 User Namespace 生效了。

2.1.7 Network Namespace

Network Namespace 是用来隔离网络设备、IP 地址端口等网络栈的 Namespace。Network Namespace 可以让每个容器拥有自己独立的（虚拟的）网络设备，而且容器内的应用可以绑定到自己的端口，每个 Namespace 内的端口都不会互相冲突。在宿主机上搭建网桥后，就能很方便地实现容器之间的通信，而且不同容器上的应用可以使用相同的端口。

同样，在 2.1.6 小节的代码的基础上增加 syscall.CLONE_NEWNET 标识符，如下。

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC | syscall.CLONE_NEWPID
|
        syscall.CLONE_NEWNS |
            syscall.CLONE_NEWUSER | syscall.CLONE_NEWNET,
    }
    cmd.SysProcAttr.Credential = &syscall.Credential{Uid: uint32(1), Gid:
    uint32(1)}
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
    os.Exit(-1)
}
```

首先，在宿主机上查看一下自己的网络设备，结果如下。

```
root@iZ254rt8xf1Z:~/gocode/src/book# ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:d7:5d:c3:b9
          inet addr:192.168.0.1  Bcast:0.0.0.0  Mask:255.255.240.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet  HWaddr 00:16:3e:00:38:cc
          inet addr:10.170.174.187  Bcast:10.170.175.255  Mask:255.255.248.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5605 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1819 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7129227 (7.1 MB)  TX bytes:159780 (159.7 KB)

eth1      Link encap:Ethernet  HWaddr 00:16:3e:00:6d:4d
          inet addr:101.200.126.205  Bcast:101.200.127.255  Mask:255.255.252.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15433 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6888 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:13287762 (13.2 MB)  TX bytes:1787482 (1.7 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

可以看到，宿主机上有 lo、eth0、eth1 等网络设备。下面，运行一下程序去 Network Namespace 里面看看。

```
root@iZ254rt8xf1Z:~/gocode/src/book# go run main.go
$ ifconfig
$
```

我们发现，在 Namespace 里面什么网络设备都没有。这样就能断定 Network Namespace 与宿主机之间的网络是处于隔离状态了。

2.2 Linux Cgroups 介绍

2.1 节讲述的是构建 Linux 容器的 Namespace 技术，它帮助进程隔离出自己单独的空间，但 Docker 是怎么限制每个空间的大小，保证它们不会互相争抢的呢？这就要用到 Linux 的 Cgroups 技术。

2.2.1 什么是 Linux Cgroups

Linux Cgroups（Control Groups）提供了对一组进程及将来子进程的资源限制、控制和统计的能力，这些资源包括 CPU、内存、存储、网络等。通过 Cgroups，可以方便地限制某个进程的资源占用，并且可以实时地监控进程的监控和统计信息。

Cgroups 中的 3 个组件

- cgroup 是对进程分组管理的一种机制，一个 cgroup 包含一组进程，并可以在这个 cgroup 上增加 Linux subsystem 的各种参数配置，将一组进程和一组 subsystem 的系统参数关联起来。
- subsystem 是一组资源控制的模块，一般包含如下几项。
  - blkio 设置对块设备（比如硬盘）输入输出的访问控制。
  - cpu 设置 cgroup 中进程的 CPU 被调度的策略。
  - cpuacct 可以统计 cgroup 中进程的 CPU 占用。
  - cpuset 在多核机器上设置 cgroup 中进程可以使用的 CPU 和内存（此处内存仅使用于 NUMA 架构）。
  - devices 控制 cgroup 中进程对设备的访问。
  - freezer 用于挂起（suspend）和恢复（resume） cgroup 中的进程。
  - memory 用于控制 cgroup 中进程的内存占用。
  - net_cls 用于将 cgroup 中进程产生的网络包分类，以便 Linux 的 tc（traffic controller） 可以根据分类区分出来自某个 cgroup 的包并做限流或监控。
  - net_prio 设置 cgroup 中进程产生的网络流量的优先级。
  - ns 这个 subsystem 比较特殊，它的作用是使 cgroup 中的进程在新的 Namespace 中 fork 新进程（NEWNS）时，创建出一个新的 cgroup，这个 cgroup 包含新的 Namespace 中的进程。

每个 subsystem 会关联到定义了相应限制的 cgroup 上，并对这个 cgroup 中的进程做相

应的限制和控制。这些 subsystem 是逐步合并到内核中的，如何看到当前的内核支持哪些 subsystem 呢？可以安装 cgroup 的命令行工具（apt-get install cgroup-bin），然后通过 lssubsys 看到 Kernel 支持的 subsystem。

```
# / lssubsys -a
cpuset
cpu,cpuacct
blkio
memory
devices
freezer
net_cls,net_prio
perf_event
hugetlb
pids
```

- hierarchy 的功能是把一组 cgroup 串成一个树状的结构，一个这样的树便是一个 hierarchy，通过这种树状结构，Cgroups 可以做到继承。比如，系统对一组定时的任务进程通过 cgroup1 限制了 CPU 的使用率，然后其中有一个定时 dump 日志的进程还需要限制磁盘 IO，为了避免限制了磁盘 IO 之后影响到其他进程，就可以创建 cgroup2，使其继承于 cgroup1 并限制磁盘的 IO，这样 cgroup2 便继承了 cgroup1 中对 CPU 使用率的限制，并且增加了磁盘 IO 的限制而不影响到 cgroup1 中的其他进程。

三个组件相互的关系

通过上面组件的描述，不难看出，Cgroups 是凭借这三个组件的相互协作实现的。那么，这三个组件是什么关系呢？

- 系统在创建了新的 hierarchy 之后，系统中所有的进程都会加入这个 hierarchy 的 cgroup 根节点，这个 cgroup 根节点是 hierarchy 默认创建的，2.2.2 小节在这个 hierarchy 中创建的 cgroup 都是这个 cgroup 根节点的子节点。
- 一个 subsystem 只能附加到一个 hierarchy 上面。
- 一个 hierarchy 可以附加多个 subsystem。
- 一个进程可以作为多个 cgroup 的成员，但是这些 cgroup 必须在不同的 hierarchy 中。
- 一个进程 fork 出子进程时，子进程是和父进程在同一个 cgroup 中的，也可以根据需要将其移动到其他 cgroup 中。

这几句话现在不理解暂时没关系，在后面实际使用的过程中会逐渐了解到它们之间的联

系的。

Kernel 接口

前面介绍了那么多 Cgroups 结构的内容，那么到底要怎么调用 Kernel 才能配置 Cgroups 呢？通过前面的介绍了解到，Cgroups 中的 hierarchy 是一种树状的组织结构，Kernel 为了使对 Cgroups 的配置更直观，是通过一个虚拟的树状文件系统配置 Cgroups 的，通过层级的目录虚拟出 cgroup 树。下面，就以一个配置的例子来了解一下如何操作 Cgroups。

1. 首先，要创建并挂载一个 hierarchy（cgroup 树），如下。

```
→ ~ mkdir cgroup-test # 创建一个 hierarchy 挂载点
→ ~ sudo mount -t cgroup -o none,name=cgroup-test cgroup-test ./cgroup-test # 挂载一个 hierarchy
→ ~ ls ./cgroup-test # 挂载后我们就可以看到系统在这个目录下生成了一些默认文件
cgroup.clone_children  cgroup.procs  cgroup.sane_behavior  notify_on_release
release_agent  tasks
```

这些文件就是这个 hierarchy 中 cgroup 根节点的配置项，上面这些文件的含义分别如下。

- cgroup.clone_children，cpuset 的 subsystem 会读取这个配置文件，如果这个值是 1（默认是 0），子 cgroup 才会继承父 cgroup 的 cpuset 的配置。
- cgroup.procs 是树中当前节点 cgroup 中的进程组 ID，现在的位置是在根节点，这个文件中会有现在系统中所有进程组的 ID。
- notify_on_release 和 release_agent 会一起使用。notify_on_release 标识当这个 cgroup 最后一个进程退出的时候是否执行了 release_agent；release_agent 则是一个路径，通常用作进程退出之后自动清理掉不再使用的 cgroup。
- tasks 标识该 cgroup 下面的进程 ID，如果把一个进程 ID 写到 tasks 文件中，便会将相应的进程加入到这个 cgroup 中。

2. 然后，创建刚刚创建好的 hierarchy 上 cgroup 根节点中扩展出的两个子 cgroup。

```
→ [cgroup-test] sudo mkdir cgroup-1 # 创建子 cgroup "cgroup-1"
→ [cgroup-test] sudo mkdir cgroup-2 # 创建子 cgroup "cgroup-2"
→ [cgroup-test] tree
.
|-- cgroup-1
|   |-- cgroup.clone_children
|   |-- cgroup.procs
|   |-- notify_on_release
|   `-- tasks
|-- cgroup-2
```

```
|   |-- cgroup.clone_children
|   |-- cgroup.procs
|   |-- notify_on_release
|   `-- tasks
|-- cgroup.clone_children
|-- cgroup.procs
|-- cgroup.sane_behavior
|-- notify_on_release
|-- release_agent
`-- tasks
```

可以看到，在一个 cgroup 的目录下创建文件夹时，Kernel 会把文件夹标记为这个 cgroup 的子 cgroup，它们会继承父 cgroup 的属性。

3. 在 cgroup 中添加和移动进程。

一个进程在一个 Cgroups 的 hierarchy 中，只能在一个 cgroup 节点上存在，系统的所有进程都会默认在根节点上存在，可以将进程移动到其他 cgroup 节点，只需要将进程 ID 写到移动到的 cgroup 节点的 tasks 文件中即可。

```
→ [cgroup-1] echo $$
7475
→ [cgroup-1] sudo sh -c "echo $$ >> tasks" # 将我所在的终端进程移动到 cgroup-1 中
→ [cgroup-1] cat /proc/7475/cgroup
13:name=cgroup-test:/cgroup-1
11:perf_event:/
10:cpu,cpuacct:/user.slice
9:freezer:/
8:blkio:/user.slice
7:devices:/user.slice
6:cpuset:/
5:hugetlb:/
4:pids:/user.slice/user-1000.slice
3:memory:/user.slice
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-1000.slice/session-19.scope
```

可以看到，当前的 7475 进程已经被加到 cgroup-test:/cgroup-1 中了。

4. 通过 subsystem 限制 cgroup 中进程的资源。

在上面创建 hierarchy 的时候，这个 hierarchy 并没有关联到任何的 subsystem，所以没办法通过那个 hierarchy 中的 cgroup 节点限制进程的资源占用，其实系统默认已经为每个 subsystem 创建了一个默认的 hierarchy，比如 memory 的 hierarchy。

```
→ ~ mount | grep memory
cgroup on /sys/fs/cgroup/memory type cgroup
(rw,nosuid,nodev,noexec,relatime,memory,nsroot=/)
```

可以看到，/sys/fs/cgroup/memory 目录便是挂在了 memory subsystem 的 hierarchy 上。下面，就通过在这个 hierarchy 中创建 cgroup，限制如下进程占用的内存。

```
→ [memory] # 首先，在不做限制的情况下，启动一个占用内存的 stress 进程
→ [memory] stress --vm-bytes 200m --vm-keep -m 1
→ [memory] # 创建一个 cgroup
→ [memory] sudo mkdir test-limit-memory && cd test-limit-memory
→ [test-limit-memory] # 设置最大 cgroup 的最大内存占用为 100MB
→ [test-limit-memory] sudo sh -c "echo "100m" > memory.limit_in_bytes"
→ [test-limit-memory] # 将当前进程移动到这个 cgroup 中
→ [test-limit-memory] sudo sh -c "echo $$ > tasks"
→ [test-limit-memory] # 再次运行占用内存 200MB 的 stress 进程
→ [test-limit-memory] stress --vm-bytes 200m --vm-keep -m 1
```

本机为 2GB 内存，运行结果如下（通过 top 命令监控）。

```
# 未限制前内存占用约 200MB (2GB * 10%)
PID  PPID     TIME+ %CPU %MEM  PR  NI S    VIRT    RES   UID COMMAND
8336  8335   0:08.23 99.0 10.0  20   0 R  212284 205060  1000 stress
8335  7475   0:00.00  0.0  0.0  20   0 S    7480    876  1000 stress

# 限制后内存占用约 100MB (2GB * 5%)
PID  PPID     TIME+ %CPU %MEM  PR  NI S    VIRT    RES   UID COMMAND
8310  8309   0:01.17  7.6  5.0  20   0 R  212284 102056  1000 stress
8309  7475   0:00.00  0.0  0.0  20   0 S    7480    796  1000 stress
```

可以看到，通过 cgroup，我们成功地将 stress 进程的最大内存占用限制到了 100MB。

2.2.2 Docker 是如何使用 Cgroups 的

我们知道 Docker 是通过 Cgroups 实现容器资源限制和监控的，下面以一个实际的容器实例来看一下 Docker 是如何配置 Cgroups 的。

```
→ ~ # docker run -m 设置内存限制
→ ~ sudo docker run -itd -m  128m ubuntu
957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11
→ ~ # docker 会为每个容器在系统的 hierarchy 中创建 cgroup
→ ~ cd
/sys/fs/cgroup/memory/docker/957459145e9092618837cf94a1cb356e206f2f0da560b40cb310
35e442d3df11
→ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 # 查看 cgroup 的
```

```
内存限制
→ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 cat memory.
limit_in_bytes
134217728
→ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 # 查看 cgroup 中
进程所使用的内存大小
→ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 cat memory.
usage_in_bytes
430080
```

可以看到，Docker 通过为每个容器创建 cgroup，并通过 cgroup 去配置资源限制和资源监控。

2.2.3　用 Go 语言实现通过 cgroup 限制容器的资源

下面，在 2.1 节中 Namespace 容器 demo 的基础上，加上 cgroup 的限制，实现一个 demo，使其能够具有限制容器内存的功能。

```
package main

import (
    "os/exec"
    "path"
    "os"
    "fmt"
    "io/ioutil"
    "syscall"
    "strconv"
)
// 挂载了 memory subsystem 的 hierarchy 的根目录位置
const cgroupMemoryHierarchyMount = "/sys/fs/cgroup/memory"

func main() {
    if os.Args[0] == "/proc/self/exe" {
        // 容器进程
        fmt.Printf("current pid %d", syscall.Getpid())
        fmt.Println()
        cmd := exec.Command("sh", "-c", `stress --vm-bytes 200m --vm-keep -m 1`)
        cmd.SysProcAttr = &syscall.SysProcAttr{
        }
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
        if err := cmd.Run(); err != nil {
            fmt.Println(err)
```

```
            os.Exit(1)
        }
    }

    cmd := exec.Command("/proc/self/exe")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_
NEWNS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Start(); err != nil {
        fmt.Println("ERROR", err)
        os.Exit(1)
    } else {
        // 得到 fork 出来进程映射在外部命名空间的 pid
        fmt.Printf("%v", cmd.Process.Pid)

        // 在系统默认创建挂载了 memory subsystem 的 Hierarchy 上创建 cgroup
        os.Mkdir(path.Join(cgroupMemoryHierarchyMount, "testmemorylimit"), 0755)
        // 将容器进程加入到这个 cgroup 中
        ioutil.WriteFile(path.Join(cgroupMemoryHierarchyMount, "testmemorylimit",
"tasks") , []byte(strconv.Itoa(cmd.Process.Pid)), 0644)
        // 限制 cgroup 进程使用
        ioutil.WriteFile(path.Join(cgroupMemoryHierarchyMount, "testmemorylimit",
"memory.limit_in_bytes") , []byte("100m"), 0644)
    }
    cmd.Process.Wait()
}
```

通过对 Cgroups 虚拟文件系统的配置，将容器中 stress 进程的内存占用限制到了 100MB（宿主机内存 2GB，5% 即 100MB）。

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
10861 root      20   0  212284 102464    212 R   6.2   5.0   0:01.13 stress
```

2.3 Union File System

2.3.1 什么是 Union File System

Union File System，简称 UnionFS，是一种为 Linux、FreeBSD 和 NetBSD 操作系统设计的，把其他文件系统联合到一个联合挂载点的文件系统服务。它使用 branch 把不同文件系统的文

件和目录“透明地”覆盖，形成一个单一一致的文件系统。这些 branch 或者是 read-only 的，或者是 read-write 的，所以当对这个虚拟后的联合文件系统进行写操作的时候，系统是真正写到了一个新的文件中。看起来这个虚拟后的联合文件系统是可以对任何文件进行操作的，但是其实它并没有改变原来的文件，这是因为 unionfs 用到了一个重要的资源管理技术，叫写时复制。

写时复制（copy-on-write，下文简称 CoW），也叫隐式共享，是一种对可修改资源实现高效复制的资源管理技术。它的思想是，如果一个资源是重复的，但没有任何修改，这时并不需要立即创建一个新的资源，这个资源可以被新旧实例共享。创建新资源发生在第一次写操作，也就是对资源进行修改的时候。通过这种资源共享的方式，可以显著地减少未修改资源复制带来的消耗，但是也会在进行资源修改时增加小部分的开销。

用一个经典的例子来解释一下。Knoppix，一个用于 Linux 演示、光盘教学和商业产品演示的 Linux 发行版，就是将一个 CD-ROM 或 DVD 和一个存在于可读写设备（比如，U 盘）上的叫作 knoppix.img 的文件系统联合起来的系统。这样，任何对 CD/DVD 上文件的改动都会被应用在 U 盘上，而不改变原来的 CD/DVD 上的内容。

2.3.2　AUFS

AUFS，英文全称是 Advanced Multi-Layered Unification Filesystem，曾经也叫 Acronym Multi-Layered Unification Filesystem、Another Multi-Layered Unification Filesystem。AUFS 完全重写了早期的 UnionFS 1.x，其主要目的是为了可靠性和性能，并且引入了一些新的功能，比如可写分支的负载均衡。AUFS 的一些实现已经被纳入 UnionFS 2.x 版本。

2.3.3　Docker 是如何使用 AUFS 的

AUFS 是 Docker 选用的第一种存储驱动。AUFS 具有快速启动容器、高效利用存储和内存的优点。直到现在，AUFS 仍然是 Docker 支持的一种存储驱动类型。接下来，介绍一下 Docker 是如何利用 AUFS 存储 image 和 container 的。

image layer 和 AUFS

每一个 Docker image 都是由一系列 read-only layer 组成的。image layer 的内容都存储在 Docker hosts filesystem 的 /var/lib/docker/aufs/diff 目录下。而 /var/lib/docker/aufs/layers 目录，则存储着 image layer 如何堆栈这些 layer 的 metadata。

准备一台安装了 Docker 1.11.2 的机器。在没有拉取任何镜像、启动任何容器的情况下，执

行如下命令。

```
ls /var/lib/docker/aufs/diff
```

可以发现，目录没有存储任何内容。

```
# 拉取 Ubuntu:15.04 镜像
$ docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
9502adfba7f1: Pull complete
4332ffb06e4b: Pull complete
2f937cc07b5f: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:2fb27e433b3ecccea2a14e794875b086711f5d49953ef173d8a03e8707f1510f
Status: Downloaded newer image for ubuntu:15.04

$ ls /var/lib/docker/aufs/diff
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77

$ ls /var/lib/docker/aufs/mnt
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77

$ cat
/var/lib/docker/aufs/layers/6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2a
d8a44e148ea
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
```

拉取镜像后，可以看到在 docker pull 中的结果显示 ubuntu:15.04 镜像一共有 4 个 layer，在执行命令的结果中也有 4 个对应的存储文件目录。

```
ls /var/lib/docker/aufs/diff
```

这里有一点需要说明的是，自从 Docker 1.10 之后，diff 目录下的存储镜像 layer 文件夹不再与镜像 ID 相同。最后，命令列出来的是堆栈里位于 6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea layer 下方的 layer。

```
cat /var/lib/docker/aufs/layers/6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2
ad8a44e148ea
```

接下来，以 ubuntu:15.04 镜像为基础镜像，创建一个名为 changed-ubuntu 的镜像。这个镜像只是在镜像的 /tmp 文件夹中添加一个写了“Hello world”的文件。可以使用下面的 Dockerfile 来实现。

```
FROM ubuntu:15.04

RUN echo "Hello world" > /tmp/newfile
```

在 terminal 中使用 cd 命令进入到上面 Dockerfile 所在的位置，执行 docker build -t changed-ubuntu. 命令来 build 镜像。

```
$docker build -t changed-ubuntu .
Sending build context to Docker daemon 10.75 kB
Step 1 : FROM ubuntu:15.04
 ---> d1b55fd07600
Step 2 : RUN echo "Hello world" > /tmp/newfile
 ---> Running in c72100f81dd1
 ---> 9d8602c9aee1
Removing intermediate container c72100f81dd1
Successfully built 9d8602c9aee1
```

然后，执行 docker images 查看现在的镜像，可以看到新生成的 changed-ubuntu 镜像。

```
$docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
changed-ubuntu      latest              9d8602c9aee1        About a minute ago  131.3 MB
ubuntu              15.04               d1b55fd07600        10 months ago       131.3 MB
```

使用如下命令，可以清楚地查看到 changed-ubuntu 镜像使用了哪些 image layer。

```
docker history changed-ubuntu
```

changed-ubuntu 镜像使用的 image layer 如下。

```
$docker history changed-ubuntu
IMAGE          CREATED         CREATED BY          SIZE                COMMENT
9d8602c9aee1   4 minutes ago   /bin/sh -c echo "Hello world" > /tmp/newfile  12 B
d1b55fd07600   10 months ago    /bin/sh -c #(nop) CMD ["/bin/bash"]          0 B
<missing>      10 months ago /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/ 1.879 kB
<missing>      10 months ago  /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic  701 B
<missing>     10 months ago  /bin/sh -c #(nop) ADD file:3f4708cf445dc1b537 131.3 MB
```

从输出中可以看到 9d8602c9aee1 image layer 位于最上层，只有 12B 的大小，由如下命令创建。

```
/bin/sh -c echo "Hello world" > /tmp/newfile
```

也就是说，changed-ubuntu 镜像只占用了 12B 的磁盘空间，这也证明了 AUFS 是如何高效使用磁盘空间的。而下面的四层 image layer，则是共享地构成 ubuntu:15.04 镜像的 4 个 image layer。“missing”标记的 layer，是自 Docker 1.10 之后，一个镜像的 image layer 的 image history 数据都存储在一个文件中导致的，这是 Docker 官方认为的正常行为。

接下来，继续查看 layer 的存储信息。

```
$ ls /var/lib/docker/aufs/diff
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e

$ ls /var/lib/docker/aufs/mnt
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e

$ cat
/var/lib/docker/aufs/layers/9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac82
8573faa3c4e
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b

$ cat
/var/lib/docker/aufs/diff/9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac8285
73faa3c4e/tmp/newfile
Hello world
```

从输出中可以看到，/var/lib/docker/aufs/diff 目录和 /var/lib/docker/aufs/mnt 目录均多了一个文件夹 9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e。当使用如下命令来查看它的 metadata 时，可以看到，它前面的 layer 就是 ubuntu:15.04 镜像所使用的 4 个 image layer。

```
cat
/var/lib/docker/aufs/layers/9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac82
8573faa3c4e
```

进一步探查 /var/lib/docker/aufs/diff/9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e 文件夹，发现其中存储了一个 /tmp/newfile 文件，文件中只有一行“Hello world”。至此，我们完整地分析出了 image layer 和 AUFS 是如何通过共享文件和文件夹来实现镜像存储的。

container layer 和 AUFS

Docker 使用 AUFS 的 CoW 技术来实现 image layer 共享和减少磁盘空间占用。CoW 意味着一旦某个文件只有很小的部分有改动，AUFS 也需要复制整个文件。这种设计会对容器性能产生一定的影响，尤其是在待复制的文件很大，或者位于很多 image layer 下方，又或者 AUFS 需要深度搜索目录结构树的时候。不过也不用过度担心，对于一个容器而言，每个 image layer 最多只需要复制一次。后续的改动都会在第一次拷贝的 container layer 上进行。

启动一个 container 的时候，Docker 会为其创建一个 read-only 的 init layer，用来存储与这个容器内环境相关的内容；Docker 还会为其创建一个 read-write 的 layer 来执行所有写操作。

container layer 的 mount 目录也是 /var/lib/docker/aufs/mnt。container 的 metadata 和配置文件都存放在 /var/lib/docker/containers/<container-id> 目录中。container 的 read-write layer 存储在 /var/lib/docker/aufs/diff/ 目录下。即使容器停止，这个可读写层仍然存在，因而重启容器不会丢失数据，只有当一个容器被删除的时候，这个可读写层才会一起删除。

接下来，仍然用实验来证明上面的结论。首先查询到现有的容器数目为 0，而且在 /var/lib/docker/containers 目录下也没有查到任何数据。最后，查看一下系统的 aufs mount 情况，发现只有一个 *config* 文件。

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
     PORTS               NAMES

$ ls /var/lib/docker/containers

$ ls /sys/fs/aufs/
```

启动一个 changed-ubuntu 的容器。

```
$docker run -dit changed-ubuntu bash
fb5939d878bb0521008d63eb06adea75e6af275855f11879dfa3992dfdaa5e3f

$docker ps -a
CONTAINER ID        IMAGE                  COMMAND             CREATED             STATUS
     PORTS               NAMES
```

```
fb5939d878bb        changed-ubuntu          "bash"      28 seconds ago   Up 27 seconds
                    amazing_babbage
```

如下，查看 /var/lib/docker/aufs/diff 目录发现，下面多了 2 个文件夹，f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8-init 是 Docker 为容器创建的 read-only 的 init layer，而 f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8 则是 Docker 为容器创建的 read-write layer。

```
$ ls /var/lib/docker/aufs/diff
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8-init
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8
```

/var/lib/docker/aufs/mnt 目录的变化和 /var/lib/docker/aufs/diff 一致。

```
$ ls /var/lib/docker/aufs/mnt
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8-init
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8
```

/var/lib/docker/aufs/layers/ 目录下多了与上文两个文件目录同名的文件，用 cat 命令可以清楚地看到其依赖 layer 的记录。

```
$ls /var/lib/docker/aufs/layers
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8-init
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8

$ cat
/var/lib/docker/aufs/layers/f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a4
2a949e2bfc8
f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8-init
```

```
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b

$ cat
/var/lib/docker/aufs/layers/f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a4
2a949e2bfc8-init
9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac828573faa3c4e
6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8a44e148ea
9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171904c17573
f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98119e2cf77
208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4a6c22b01b
```

在 /var/lib/docker/containers/ 目录下多了一个与 containerid 相同的文件夹，存放着容器的 metadata 和 config 文件。

```
$ ls /var/lib/docker/containers/
fb5939d878bb0521008d63eb06adea75e6af275855f11879dfa3992dfdaa5e3f

$ ls
/var/lib/docker/containers/fb5939d878bb0521008d63eb06adea75e6af275855f11879dfa399
2dfdaa5e3f/
config.v2.json   fb5939d878bb0521008d63eb06adea75e6af275855f11879dfa3992dfdaa5e3f-
json.log  hostconfig.json  hostname  hosts  resolv.conf  resolv.conf.hash  shm
```

接下来从系统 AUFS 来看 mount 的情况，在 /sys/fs/aufs/ 目录下多了一个 si_fe6d5733e85e4904 文件夹，执行如下命令。

```
cat /sys/fs/aufs/si_fe6d5733e85e4904/*
```

下面，可以清楚地看到这就是刚刚创建的容器的 layer 权限，只有最上面的 f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949e2bfc8 layer 是 read-write 权限。

```
$ls /sys/fs/aufs/
config  si_fe6d5733e85e4904

$ cat /sys/fs/aufs/si_fe6d5733e85e4904/*
/var/lib/docker/aufs/diff/f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a949
e2bfc8=rw
/var/lib/docker/aufs/diff/f9ccf5caa9b7324f0ef112750caa14203b557d276ca08c78c23a42a
949e2bfc8-init=ro+wh
/var/lib/docker/aufs/diff/9f122dbaa103338f27bac146326af38a2bcb52f98ebb3530cac8285
```

```
73faa3c4e=ro+wh
/var/lib/docker/aufs/diff/6bb19cb345da470e015ba3f1ca049a1c27d2c57ebc205ec165d2ad8
a44e148ea=ro+wh
/var/lib/docker/aufs/diff/9c444e426a4a0aa3ad8ff162dd7bcd4dcbb2e55bdec268b24666171
904c17573=ro+wh
/var/lib/docker/aufs/diff/f193107618deb441376a54901bc9115f30473c1ec792b7fb3e73a98
119e2cf77=ro+wh
/var/lib/docker/aufs/diff/208319b22189a2c3841bc4a4ef0df9f9238a3e832dc403133fb8ad4
a6c22b01b=ro+wh
64
65
66
67
68
69
70
/run/shm/aufs.xino
```

最后，讲一下 AUFS 如何为 container 删除一个文件。如果要删除 file1，AUFS 会在 container 的 read-write 层生成一个 .wh.file1 的文件来隐藏所有 read-only 层的 file1 文件。至此，我们已清楚地描述和验证了 Docker 是如何使用 AUFS 来管理 container layer 的。

2.3.4 自己动手写 AUFS

下面，开始自己动手用简单的命令来创建一个 AUFS 文件系统，感受下如何使用 AUFS 和 CoW 实现文件管理。

首先，在实验目录下创建一个 aufs 的文件夹，然后在 aufs 目录下创建一个 mnt 的文件夹作挂载点。接着，在 aufs 目录下创建一个名为 container-layer 的文件夹，里面有一个名为 container-layer.txt 的文件，文件内容为“I am container layer”。同样地，继续在 aufs 目录下创建 4 个名为 image-layern 的文件夹（n 取值分别为 1 和 4），里面有一个名为 image-layer{n}.txt 的文件，文件内容为“I am image layer${n}”。使用如下命令检查文件内容。

```
$ cd /home/qinyujia/aufs

$ ls
container-layer  image-layer1  image-layer2  image-layer3  image-layer4  mnt

$ cat container-layer.txt
I am container layer

$ cat image-layer1/image-layer1.txt
```

```
I am image layer 1

$ cat image-layer2/image-layer2.txt
I am image layer 2

$ cat image-layer3/image-layer3.txt
I am image layer 3

$ cat image-layer4/image-layer4.txt
I am image layer 4
```

要联合的文件目录都已经准备好了。接下来，把 container-layer 和 4 个名为 image-layer${n} 的文件夹用 AUFS 的方式挂载到刚刚创建的 mnt 目录下。在 mount aufs 的命令中，没有指定待挂载的 5 个文件夹的权限，默认的行为是，dirs 指定的左边起第一个目录是 read-write 权限，后续的都是 read-only 权限。

```
$ sudo mount -t aufs -o dirs=./container-layer:./image-layer4:./image-layer3:./image-
layer2:./image-layer1 none ./mnt

$ tree mnt
mnt
├── container-layer.txt
├── image-layer1.txt
├── image-layer2.txt
├── image-layer3.txt
└── image-layer4.txt
```

大家还记得上一小节曾经在系统 aufs 目录下，查看文件读写权限的做法吗？这里依然使用如下命令来确认新 mount 的文件系统中每个目录的权限。（注意，si_fe6d5733e85e5904 应该是系统为这个 mnt 挂载点新创建的，而非在介绍 Docker 和 AUFS 里面提到的那个文件夹。）

```
cat /sys/fs/aufs/si_fe6d5733e85e5904/*
```

根据输出，可以清楚地看到，只有 container-layer 文件夹是 read-write 的，其余的都是 read-only 权限。

```
$ cat /sys/fs/aufs/si_fe6d5733e85e5904/*
/home/qinyujia/aufs/container-layer=rw
/home/qinyujia/aufs/image-layer4=ro
/home/qinyujia/aufs/image-layer3=ro
/home/qinyujia/aufs/image-layer2=ro
/home/qinyujia/aufs/image-layer1=ro
64
65
```

```
66
67
68
/home/qinyujia/aufs/container-layer/.aufs.xino
```

接下来，执行一个有意思的操作，往 mnt/image-layer1.txt 文件末尾添加一行文字“write to mnt's image-layer1.txt”。根据上面介绍的 CoW 技术，大家猜想一下会产生什么样的行为。

```
$ echo -e "\nwrite to mnt's image-layer1.txt" >> ./mnt/image-layer4.txt
```

用 cat 命令去查看 mnt/image-layer4.txt 文件的内容，发现内容确实从“I am image layer 4”变成了如下的样子。

```
“I am image layer 4

write to mnt's image-layer1.txt”
```

此处，mnt 只是一个虚拟挂载点，因此，接下来还需要继续去寻找文件修改到底在什么位置。

```
$ cat ./mnt/image-layer4.txt
I am image layer 4

write to mnt's image-layer1.txt
```

查看 image-layer4/image-layer4.txt 文件的内容，发现其并未改变。

```
$ cat image-layer4/image-layer4.txt
I am image layer 4
```

而在检查 container-layer 文件夹的时候，发现多了一个名为 image-layer4.txt 的文件，文件的内容如下。

```
"I am image layer 4

write to mnt's image-layer1.txt"
```

也就是说，当尝试向 mnt/image-layer4.txt 文件进行写操作的时候，系统首先在 mnt 目录下查找名为 image-layer4.txt 的文件，将其拷贝到 read-write 层的 container-layer 目录中，接着对 container-layer 目录中的 image-layer4.txt 文件进行写操作。至此，我们成功地完成了一个小小的 demo，实现了自己的 AUFS 文件系统。

```
$ ls container-layer/
container-layer.txt  image-layer4.txt
```

```
$cat container-layer/image-layer4.txt
I am image layer 4

write to mnt's image-layer1.txt
```

2.4　小结

本章首先介绍了 Linux Namespace，一共有 6 种类别的 Namespace，分别进行了简单介绍。然后，以 Go 语言为例实现了一个 demo，使大家能有一个直观的认识。在第 3 章中会使用到这些知识，而且对于这些 namespace 的应用，会有更加复杂的例子等待着大家。

2.2 节介绍了 Linux Cgroups。通过 Linux Cgroups 的三种结构，可以随意定制对资源的限制及对资源做监控。最后，使用 Go 语言实现了一个 Cgroups 限制资源的 demo，介绍了如何用 Go 语言去操控容器的 Cgroups，进而实现限制容器资源的效果。

2.3 节介绍了 Union File System。列举了其中的几个具体实现，并且讲解了 Docker 是如何使用分层文件系统来实现镜像不同分层的重复利用的。最后，以 AUFS 为例子介绍了如何构建一个简单的分层文件系统。后面在开发自己的容器镜像的过程中就会使用这项技术。

第3章 构造容器

3.1 构造实现 run 命令版本的容器

本节代码获取方式：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-3.1
```

本章即将开始真正踏上构造自己的容器的道路。我们会基于当前的操作系统创建一个与宿主机隔离的容器环境，下面就开始吧。

3.1.1 Linux proc 文件系统介绍

在开始之前，还是需要稍微补充一些基本知识。但如果你对这些基本知识已经很熟悉了，请直接略过。Linux 下的 /proc 文件系统是由内核提供的，它其实不是一个真正的文件系统，只包含了系统运行时的信息（比如系统内存、mount 设备信息、一些硬件配置等），它只存在于内存中，而不占用外存空间。它以文件系统的形式，为访问内核数据的操作提供接口。实际上，很多系统工具都是简单地去读取这个文件系统的某个文件内容，比如 lsmod，其实就是 cat /proc/modules。

当遍历这个目录的时候，会发现很多数字，这些都是为每个进程创建的空间，数字就是它们的 PID。

```
:~# ls /proc/
1     1216  1320  154  20006  22    27462 32    37  44  51  60  78  868 acpi
devices     interrupts keys          meminfo      sched_debug sys
```

```
version_signature
10    12180 1336  155  20007  23    27478 32025 38  45  52  61  780 896 buddyinfo
diskstats   iomem       kmsg          misc          schedstat   sysrq-trigger
vmallocinfo
100   12192 13567 16   20008  24    28    33    387 47  53  62  8   9   bus   dma
ioports     kpagecount modules       scsi          sysvipc     vmstat
10684 12728 14    17   20011  25    29    337   39  48  54  63  818 944 cgroups
driver      ipmi       kpageflags    mounts        self        timer_list    xen
10698 1279  145   18   20012  26    3     34    40  49  55  7   859 954 cmdline
execdomains irq        latency_stats mtrr          slabinfo    timer_stats
zoneinfo
11    13    15    19   20013  27    30    343   41  5   57  715 861 961 consoles
fb          kallsyms   loadavg       net           softirqs    tty
1170  1301  1505  2    20042  27396 31    35    42  50  58  75  864 99  cpuinfo
filesystems  kcore     locks         pagetypeinfo  stat        uptime
12    13131 1508  20   21     27443 31809 36    43  501 59  77  865 991 crypto
fs          key-users  mdstat        partitions    swaps       version
```

下面介绍几个比较重要的部分。

/proc/N	PID 为 N 的进程信息
/proc/N/cmdline	进程启动命令
/proc/N/cwd	链接到进程当前工作目录
/proc/N/environ	进程环境变量列表
/proc/N/exe	链接到进程的执行命令文件
/proc/N/fd	包含进程相关的所有文件描述符
/proc/N/maps	与进程相关的内存映射信息
/proc/N/mem	指代进程持有的内存，不可读
/proc/N/root	链接到进程的根目录
/proc/N/stat	进程的状态
/proc/N/statm	进程使用的内存状态
/proc/N/status	进程状态信息，比 stat/statm 更具可读性
/proc/self/	链接到当前正在运行的进程

3.1.2　实现 run 命令

首先实现一个简单版本的 run 命令，类似于 docker run -ti [command]。然后在 3.2 节到 3.4 节中逐步添加 network、mount filesystem 等功能。为了方便了解 Docker 启动容器的原理，该简单版本的实现参考 runC 的实现。

目前的代码文件结构如下。

```
mydocker
    ├── container
```

```
|   ├── container_process.go
|   └── init.go
├── Godeps
|   ├── Godeps.json
|   └── Readme
├── main_command.go
├── main.go
├── mydocker
├── run.go
└── vendor
```

首先，来看一下入口 main 文件。

```
package main

import (
    log "github.com/Sirupsen/logrus"
    "github.com/urfave/cli"
    "os"
)

const usage = `mydocker is a simple container runtime implementation.
               The purpose of this project is to learn how docker works and how to
write a docker by ourselves
               Enjoy it, just for fun.`

func main() {
    app := cli.NewApp()
    app.Name = "mydocker"
    app.Usage = usage

    app.Commands = []cli.Command{
        initCommand,
        runCommand,
    }

    app.Before = func(context *cli.Context) error {
        // Log as JSON instead of the default ASCII formatter.
        log.SetFormatter(&log.JSONFormatter{})

        log.SetOutput(os.Stdout)
        return nil
    }

    if err := app.Run(os.Args); err != nil {
        log.Fatal(err)
    }
}
```

使用 github.com/urfave/cli 提供的命令行工具，定义了 mydocker 的几个基本命令，包括 runCommand 和 initCommand，然后在 app.Before 内初始化一下 logrus 的日志配置。下面来看一下子命令的具体定义。

```
// 这里定义了 runCommand 的 Flags，其作用类似于运行命令时使用 -- 来指定参数
var runCommand = cli.Command{
    Name:  "run",
    Usage: `Create a container with namespace and cgroups limit
           mydocker run -ti [command]`,
    Flags: []cli.Flag{
        cli.BoolFlag{
            Name:        "ti",
            Usage:       "enable tty",
        },
    },
    /*
    这里是 run 命令执行的真正函数。
    1. 判断参数是否包含 command
    2. 获取用户指定的 command
    3. 调用 Run function 去准备启动容器
    */
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Missing container command")
        }
        cmd := context.Args().Get(0)
        tty := context.Bool("ti")
        Run(tty, cmd)
        return nil
    },
}

// 这里，定义了 initCommand 的具体操作，此操作为内部方法，禁止外部调用
var initCommand = cli.Command{
    Name:   "init",
    Usage:  "Init container process run user's process in container. Do not call
            it outside",
    /*
    1. 获取传递过来的 command 参数
    2. 执行容器初始化操作
    */
    Action: func(context *cli.Context) error {
        log.Infof("init come on")
        cmd := context.Args().Get(0)
```

```
            log.Infof("command %s", cmd)
            err := container.RunContainerInitProcess(cmd, nil)
            return err
        },
    }
```

先来看一下 run 函数做了哪些事情。

```
/*
这里是父进程，也就是当前进程执行的内容，根据上一章介绍的内容，应该比较容易明白。
1. 这里的 /proc/self/exe 调用中，/proc/self/ 指的是当前运行进程自己的环境，exec 其实就是自己调用了自己，使用这种方式对创建出来的进程进行初始化
2. 后面的 args 是参数，其中 init 是传递给本进程的第一个参数，在本例中，其实就是会去调用 initCommand 去初始化进程的一些环境和资源
3. 下面的 clone 参数就是去 fork 出来一个新进程，并且使用了 namespace 隔离新创建的进程和外部环境。
4. 如果用户指定了 -ti 参数，就需要把当前进程的输入输出导入到标准输入输出上
*/
func NewParentProcess(tty bool, command string) *exec.Cmd {
    args := []string{"init", command}
    cmd := exec.Command("/proc/self/exe", args...)
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_
        NEWNS | syscall.CLONE_NEWNET | syscall.CLONE_NEWIPC,
    }
    if tty {
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
    }
    return cmd
}

/*
这里的 Start 方法是真正开始前面创建好的 command 的调用，它首先会 clone 出来一个 namespace 隔离的进程，然后在子进程中，调用 /proc/self/exe，也就是调用自己，发送 init 参数，调用我们写的 init 方法，去初始化容器的一些资源。
*/
func Run(tty bool, command string) {
    parent := container.NewParentProcess(tty, command)
    if err := parent.Start(); err != nil {
        log.Error(err)
    }
    parent.Wait()
    os.Exit(-1)
}
```

那么，init 函数里面做了些什么呢？

```
/*
这里的 init 函数是在容器内部执行的，也就是说，代码执行到这里后，容器所在的进程其实就已经创建出来了，
这是本容器执行的第一个进程。
使用 mount 先去挂载 proc 文件系统，以便后面通过 ps 等系统命令去查看当前进程资源的情况。
*/
func RunContainerInitProcess(command string, args []string) error {
    logrus.Infof("command %s", command)

    defaultMountFlags := syscall.MS_NOEXEC | syscall.MS_NOSUID | syscall.MS_NODEV
    syscall.Mount("proc", "/proc", "proc", uintptr(defaultMountFlags), "")
    argv := []string{command}
    if err := syscall.Exec(command, argv, os.Environ()); err != nil {
        logrus.Errorf(err.Error())
    }
    return nil
}
```

这里的 MountFlag 的意思如下。

- MS_NOEXEC 在本文件系统中不允许运行其他程序。
- MS_NOSUID 在本系统中运行程序的时候，不允许 set-user-ID 或 set-group-ID。
- MS_NODEV 这个参数是自从 Linux 2.4 以来，所有 mount 的系统都会默认设定的参数。

本函数最后的 syscall.Exec，是最为重要的一句黑魔法，正是这个系统调用实现了完成初始化动作并将用户进程运行起来的操作。你可能会说，这有什么神奇的，不就是运行一下程序嘛。下面来解释一下这句话的神奇之处。

首先，使用 Docker 创建起来一个容器之后，会发现容器内的第一个程序，也就是 PID 为 1 的那个进程，是指定的前台进程。那么，根据 3.1.1 小节所讲的过程发现，容器创建之后，执行的第一个进程并不是用户的进程，而是 init 初始化的进程。这时候，如果通过 ps 命令查看就会发现，容器内第一个进程变成了自己的 init，这和预想的是不一样的。你可能会想，大不了把第一个进程给 kill 掉。但这里又有一个令人头疼的问题，PID 为 1 的进程是不能被 kill 掉的，如果该进程被 kill 掉，我们的容器也就退出了。那么，有什么办法呢？这里的 execve 系统调用就可以大显神威了。

syscall.Exec 这个方法，其实最终调用了 Kernel 的 int execve(const char *filename, char *const argv[], char *const envp[]); 这个系统函数。它的作用是执行当前 filename 对应的程序。它会覆盖当前进程的镜像、数据和堆栈等信息，包括 PID，这些都会被将要运行的进程覆盖掉。

也就是说，调用这个方法，将用户指定的进程运行起来，把最初的 init 进程给替换掉，这样当进入到容器内部的时候，就会发现容器内的第一个程序就是我们指定的进程了。这其实也是目前 Docker 使用的容器引擎 runC 的实现方式之一。

流程图如图 3.1 所示。

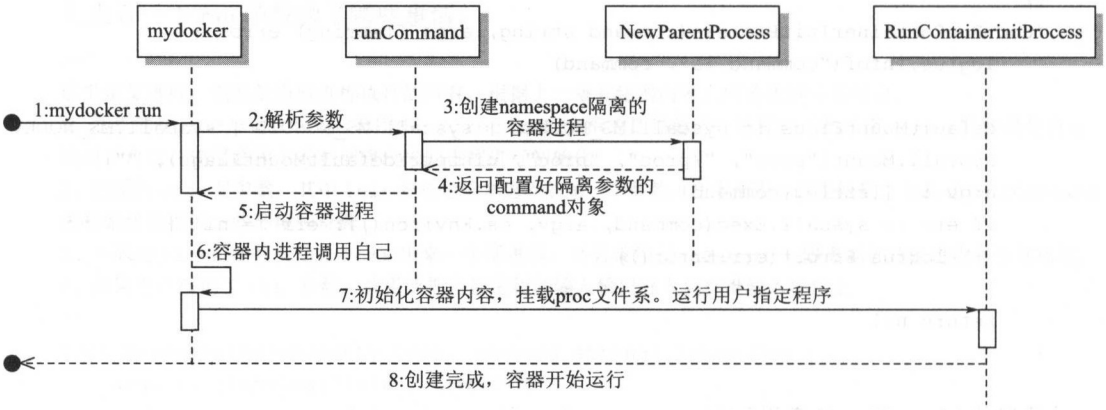


图 3.1

到这里，代码就差不多讲完了。下面来将其编译运行一下。

```
# 使用 go build . 在 mydocker 目录下进行编译
root@ubuntu:[mydocker]# go build .
# 使用 ./mydocker run -ti /bin/sh 命令，其中 -ti 表示想要以交互式的形式运行容器，/bin/sh 为指定容器内运行的第一个进程
root@ubuntu:[mydocker]# ./mydocker run -ti /bin/sh
{"level":"info","msg":"init come on","time":"2016-11-27T06:15:44Z"}
{"level":"info","msg":"command /bin/sh","time":"2016-11-27T06:15:44Z"}
# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 06:15 pts/0    00:00:00 /bin/sh
root         4     1  0 06:16 pts/0    00:00:00 ps -ef
```

在容器运行 ps -ef 时，可以发现 /bin/sh 进程是容器内的第一个进程，PID 为 1，而 ps -ef 是 PID 为 1 的父进程创建出来的。来对比一下 Docker 运行的容器的效果，如下。

```
root@iZ254rt8xf1Z:~# docker run -ti ubuntu /bin/sh
# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  1 06:25 ?        00:00:00 /bin/sh
root         5     1  0 06:25 ?        00:00:00 ps -ef
```

是不是有些类似呢？这里的 /bin/sh 是一个会在前台一直运行的进程，那么可以试一下如果指定一个运行完就会退出的进程会是什么效果。

```
root@ubuntu:[mydocker]# ./mydocker run -ti /bin/ls
{"level":"info","msg":"init come on","time":"2016-11-27T06:28:31Z"}
{"level":"info","msg":"command /bin/ls","time":"2016-11-27T06:28:31Z"}
container  Godeps  main_command.go  main.go  mydocker  run.go  vendor
```

由于没有 chroot，所以目前的系统文件系统是继承自父进程的。运行了一下 ls 命令，发现容器启动起来以后，打印出了当前目录的内容，然后便退出了，这个结果和 Docker 要求容器必须有一个一直在前台运行的进程的要求一致。

3.2　增加容器资源限制

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-3.2
```

上一节中，已经可以通过命令行 mydocker run -ti 的方式创建并启动容器。这一节，将通过 cgroup 对容器的资源进行控制。

这一节中，将实现通过 mydocker run -ti -m 100m -cpuset 1 -cpushare 512 /bin/sh 的方式控制容器的内存和 CPU 配置。

3.2.1　定义 Cgroups 的数据结构

上一章中，对 Cgroups 包含的 3 个概念进行了介绍，这里做如下简单回顾。

- cgroup hierarchy 中的节点，用于管理进程和 subsystem 的控制关系。
- subsystem 作用于 hierarchy 上的 cgroup 节点，并控制节点中进程的资源占用。
- hierarchy 将 cgroup 通过树状结构串起来，并通过虚拟文件系统的方式暴露给用户。

根据上面 3 个概念的关系，先创建出如下的数据结构。

```
package subsystems

// 用于传递资源限制配置的结构体，包含内存限制，CPU 时间片权重，CPU 核心数
type ResourceConfig struct {
    MemoryLimit string
    CpuShare    string
    CpuSet      string
```

```
}
// Subsystem 接口，每个 Subsystem 可以实现下面的 4 个接口
// 这里将 cgroup 抽象成了 path，原因是 cgroup 在 hierarchy 的路径，便是虚拟文件系统中的虚拟路径
type Subsystem interface {
    // 返回 subsystem 的名字，比如 cpu memory。
    Name() string
    // 设置某个 cgroup 在这个 Subsystem 中的资源限制
    Set(path string, res *ResourceConfig) error
    // 将进程添加到某个 cgroup 中
    Apply(path string, pid int) error
    // 移除某个 cgroup
    Remove(path string) error
}

// 通过不同的 subsystem 初始化实例创建资源限制处理链数组
var (
    SubsystemsIns = []Subsystem{
        &CpusetSubSystem{},
        &MemorySubSystem{},
        &CpuSubSystem{},
    }
)
```

上面定义了 subsystem 的模型，下面以 memory 的 subsystem 为例介绍一下要怎么实现 subsystem 的操作。

```
package subsystems

import(
    "fmt"
    "io/ioutil"
    "os"
    "path"
    "strconv"
)
// memory subsystem 的实现
type MemorySubSystem struct {
}

// 设置 cgroupPath 对应的 cgroup 的内存资源限制
func (s *MemorySubSystem) Set(cgroupPath string, res *ResourceConfig) error {
    /*
    GetCgroupPath 的作用是获取当前 subsystem 在虚拟文件系统中的路径，GetCgroupPath 这个
    函数在下面会介绍。
```

```
    */
    if subsysCgroupPath, err := GetCgroupPath(s.Name(),cgroupPath, true); err == nil {
        if res.MemoryLimit != "" {
            /*
            设置这个 cgroup 的内存限制，即将限制写入到 cgroup 对应目录的 memory.limit_in_bytes
            文件中。
            */
            if err := ioutil.WriteFile(path.Join(subsysCgroupPath, "memory.limit_
in_bytes"), []byte(res.MemoryLimit), 0644); err != nil {
                return fmt.Errorf("set cgroup memory fail %v", err)
            }
        }
        return nil
    } else {
        return err
    }

}
// 删除 cgroupPath 对应的 cgroup
func (s *MemorySubSystem) Remove(cgroupPath string) error {
    if subsysCgroupPath, err := GetCgroupPath(s.Name(), cgroupPath, false); err == nil {
        // 删除 cgroup 便是删除对应的 cgroupPath 的目录
        return os.Remove(subsysCgroupPath)
    } else {
        return err
    }
}

// 将一个进程加入到 cgroupPath 对应的 cgroup 中
func (s *MemorySubSystem) Apply(cgroupPath string, pid int) error {
    if subsysCgroupPath, err := GetCgroupPath(s.Name(), cgroupPath, false); err == nil {
        // 把进程的 PID 写到 cgroup 的虚拟文件系统对应目录下的 "task" 文件中
        if err := ioutil.WriteFile(path.Join(subsysCgroupPath, "tasks"), []
byte(strconv.Itoa(pid)), 0644); err != nil {
            return fmt.Errorf("set cgroup proc fail %v", err)
        }
        return nil
    } else {
        return fmt.Errorf("get cgroup %s error: %v", cgroupPath, err)
    }
}

// 返回 cgroup 的名字
func (s *MemorySubSystem) Name() string {
```

```
    return "memory"
}
```

上面以 memory 的 subsystem 为例，介绍了如何实现 subsystem 的 cgroup 资源限制。其中，GetCgroupPath 函数是找到对应 subsystem 挂载的 hierarchy 相对路径对应的 cgroup 在虚拟文件系统中的路径，然后通过这个目录的读写去操作 cgroup。那么，是如何找到挂载了 subsystem 的 hierarchy 的挂载目录的呢？先来熟悉下“/proc/{pid}/mountinfo”文件，如下。

```
→  ~ cat /proc/self/mountinfo
.
.
.
24 18 0:12 / /sys/kernel/security rw,nosuid,nodev,noexec,relatime shared:8 -
securityfs securityfs rw
25 20 0:19 / /dev/shm rw,nosuid,nodev shared:4 - tmpfs tmpfs rw
26 22 0:20 / /run/lock rw,nosuid,nodev,noexec,relatime shared:6 - tmpfs tmpfs
rw,size=5120k
27 18 0:21 / /sys/fs/cgroup ro,nosuid,nodev,noexec shared:9 - tmpfs tmpfs
ro,mode=755
28 27 0:22 / /sys/fs/cgroup/systemd rw,nosuid,nodev,noexec,relatime shared:10
- cgroup cgroup rw,xattr,release_agent=/lib/systemd/systemd-cgroups-
agent,name=systemd
29 18 0:23 / /sys/fs/pstore rw,nosuid,nodev,noexec,relatime shared:11 - pstore
pstore rw
30 27 0:24 / /sys/fs/cgroup/memory rw,nosuid,nodev,noexec,relatime shared:13 -
cgroup cgroup rw,memory
31 27 0:25 / /sys/fs/cgroup/freezer rw,nosuid,nodev,noexec,relatime shared:14 -
cgroup cgroup rw,freezer
32 27 0:26 / /sys/fs/cgroup/hugetlb rw,nosuid,nodev,noexec,relatime shared:15 -
cgroup cgroup rw,hugetlb
33 27 0:27 / /sys/fs/cgroup/blkio rw,nosuid,nodev,noexec,relatime shared:16 -
cgroup cgroup rw,blkio
34 27 0:28 / /sys/fs/cgroup/devices rw,nosuid,nodev,noexec,relatime shared:17 -
cgroup cgroup rw,devices
35 27 0:29 / /sys/fs/cgroup/perf_event rw,nosuid,nodev,noexec,relatime shared:18
- cgroup cgroup rw,perf_event
36 27 0:30 / /sys/fs/cgroup/pids rw,nosuid,nodev,noexec,relatime shared:19 -
cgroup cgroup rw,pids
37 27 0:31 / /sys/fs/cgroup/net_cls,net_prio rw,nosuid,nodev,noexec,relatime
shared:20 - cgroup cgroup rw,net_cls,net_prio
38 27 0:32 / /sys/fs/cgroup/cpu,cpuacct rw,nosuid,nodev,noexec,relatime shared:21
- cgroup cgroup rw,cpu,cpuacct
39 27 0:33 / /sys/fs/cgroup/cpuset rw,nosuid,nodev,noexec,relatime shared:22 -
cgroup cgroup rw,cpuset
40 19 0:34 / /proc/sys/fs/binfmt_misc rw,relatime shared:23 - autofs systemd-1 rw
```

```
,fd=22,pgrp=1,timeout=0,minproto=5,maxproto=5,direct
.
.
.
```

通过 /proc/self/mountinfo，可以找出与当前进程相关的 mount 信息。在上一章中，我们讲过 Cgroups 的 hierarchy 的虚拟文件系统是通过 cgroup 类型文件系统的 mount 挂载上去的，option 中加上 subsystem，代表挂载的 subsystem 类型，这样就可以在 mountinfo 中找到对应的 subsystem 的挂载目录了，比如 memory。

```
30 27 0:24 / /sys/fs/cgroup/memory rw,nosuid,nodev,noexec,relatime shared:13 -
cgroup cgroup rw,memory
```

通过最后的 option 是 rw,memory，可以看出这一条挂载的 subsystem 是 memory，那么在 /sys/fs/cgroup/memory 中创建文件夹对应创建的 cgroup，就可以用来做内存的限制，实现如下。

```
// 通过 /proc/self/mountinfo 找出挂载了某个 subsystem 的 hierarchy cgroup 根节点所在的目录
FindCgroupMountpoint("memory")
func FindCgroupMountpoint(subsystem string) string {
    f, err := os.Open("/proc/self/mountinfo")
    if err != nil {
        return ""
    }
    defer f.Close()

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        txt := scanner.Text()
        fields := strings.Split(txt, " ")
        for _, opt := range strings.Split(fields[len(fields)-1], ",") {
            if opt == subsystem {
                return fields[4]
            }
        }
    }
    if err := scanner.Err(); err != nil {
        return ""
    }

    return ""
}

// 得到 cgroup 在文件系统中的绝对路径
func GetCgroupPath(subsystem string, cgroupPath string, autoCreate bool) (string, error) {
```

```
    cgroupRoot := FindCgroupMountpoint(subsystem)
    if _, err := os.Stat(path.Join(cgroupRoot, cgroupPath)); err == nil ||
(autoCreate && os.IsNotExist(err)) {
        if os.IsNotExist(err) {
            if err := os.Mkdir(path.Join(cgroupRoot, cgroupPath), 0755); err ==
            nil {
            } else {
                return "", fmt.Errorf("error create cgroup %v", err)
            }
        }
        return path.Join(cgroupRoot, cgroupPath), nil
    } else {
        return "", fmt.Errorf("cgroup path error %v", err)
    }
}
```

最后，需要把这些不同 subsystem 中的 cgroup 管理起来，并与容器建立关系。

```
type CgroupManager struct {
    // cgroup 在 hierarchy 中的路径，相当于创建的 cgroup 目录相对于各 root cgroup 目录的路径
    Path     string
    // 资源配置
    Resource *subsystems.ResourceConfig
}

func NewCgroupManager(path string) *CgroupManager {
    return &CgroupManager{
        Path: path,
    }
}

// 将进程 PID 加入到每个 cgroup 中
func (c *CgroupManager) Apply(pid int) error {
    for _, subSysIns := range(subsystems.SubsystemsIns) {
        subSysIns.Apply(c.Path, pid)
    }
    return nil
}

// 设置各个 subsystem 挂载中的 cgroup 资源限制
func (c *CgroupManager) Set(res *subsystems.ResourceConfig) error {
    for _, subSysIns := range(subsystems.SubsystemsIns) {
        subSysIns.Set(c.Path, res)
    }
    return nil
```

```
}

// 释放各个 subsystem 挂载中的 cgroup
func (c *CgroupManager) Destroy() error {
    for _, subSysIns := range(subsystems.SubsystemsIns) {
        if err := subSysIns.Remove(c.Path); err != nil {
            logrus.Warnf("remove cgroup fail %v", err)
        }
    }
    return nil
}
```

通过 CgroupManager，将资源限制的配置，以及将进程移动到 cgroup 中的操作交给各个 subsystem 去处理。

如下，用一张流程图（图 3.2）来展示下上各个组件之间的调用关系。

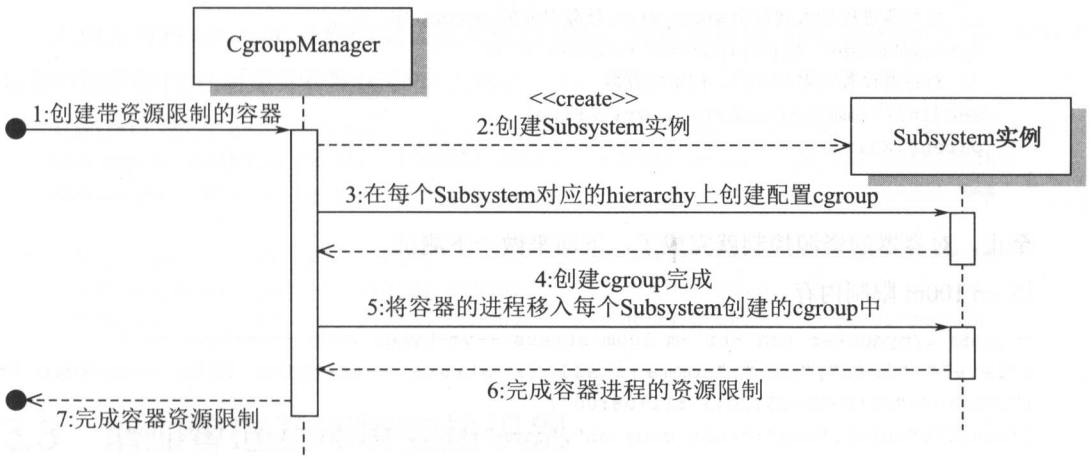


图 3.2

如图 3.2，CgroupManager 在配置容器资源限制时，首先会初始化 Subsystem 的实例，然后遍历调用 Subsystem 实例的 Set 方法，创建和配置不同 Subsystem 挂载的 hierarchy 中的 cgroup，最后再通过调用 Subsystem 实例将容器的进程分别加入到那些 cgroup 中，实现容器的资源限制。

3.2.2　在启动容器时增加资源限制的配置

在上一节中，介绍了 github.com/urfave/cli 的工具可以方便地创建出命令行的程序。现在，把资源限制的标签也加上，并在容器创建出来初始化之后，将容器的进程加到各 Subsystem 挂

载的 cgroup 中。

```
func Run(tty bool, comArray []string, res *subsystems.ResourceConfig) {
    parent, writePipe := container.NewParentProcess(tty)
    if parent == nil {
        log.Errorf("New parent process error")
        return
    }
    if err := parent.Start(); err != nil {
        log.Error(err)
    }
    // use mydocker-cgroup as cgroup name
    // 创建 cgroup manager，并通过调用 set 和 apply 设置资源限制并使限制在容器上生效
    cgroupManager := cgroups.NewCgroupManager("mydocker-cgroup")
    defer cgroupManager.Destroy()
    // 设置资源限制
    cgroupManager.Set(res)
    // 将容器进程加入到各个 subsystem 挂载对应的 cgroup 中
    cgroupManager.Apply(parent.Process.Pid)
    // 对容器设置完限制之后，初始化容器
    sendInitCommand(comArray, writePipe)
    parent.Wait()
}
```

至止，对容器的资源控制就完成了，下面来做一下测试。

用 -m 100m 限制内存。

```
→ sudo ./mydocker run -ti -m 100m stress --vm-bytes 200m --vm-keep -m 1
{"level":"info","msg":"command all is stress --vm-bytes 200m --vm-keep -m
1","time":"2016-12-03T20:37:11+08:00"}
{"level":"info","msg":"init come on","time":"2016-12-03T20:37:11+08:00"}
{"level":"info","msg":"Find path /usr/bin/stress","time":"2016-12-
03T20:37:11+08:00"}
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
```

可以看到占用内存被限制到了 100MB（宿主机内存 2GB，内存占用 5%）。

```
top - 20:41:49 up 16:28,  3 users,  load average: 1.18, 0.96, 0.43
Tasks: 122 total,   2 running, 120 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.4 us,  5.3 sy,  0.0 ni,  0.0 id, 90.4 wa,  0.0 hi,  3.9 si,  0.0 st
KiB Mem :  2048416 total,  1592804 free,   171912 used,   283700 buff/cache
KiB Swap:  2097148 total,  1892232 free,   204916 used.  1715080 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 2874 root      20   0  212284 102352    276 R  7.6  5.0   0:03.84 stress
```

用 -cpushare 512 限制一下 CPU 的时间片分配比例。启动两个占用 CPU 的 stress 进程，一个容器设置 -cpushare 512。

```
→ nohup stress --vm-bytes 200m --vm-keep -m 1 &
[1] 3000
nohup: ignoring input and appending output to 'nohup.out'
→ sudo ./mydocker run -ti -cpushare 512 stress --vm-bytes 200m --vm-keep -m 1
{"level":"info","msg":"command all is stress --vm-bytes 200m --vm-keep -m
1","time":"2016-12-03T21:00:28+08:00"}
{"level":"info","msg":"init come on","time":"2016-12-03T21:00:28+08:00"}
{"level":"warning","msg":"stress --vm-bytes 200m --vm-keep -m 1","time":"2016-12-
03T21:00:28+08:00"}
{"level":"info","msg":"Find path /usr/bin/stress","time":"2016-12-
03T21:00:28+08:00"}
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
```

上面没有将 cpushare 进程的默认值设置为 1024。通过设置容器的 -cpushare 512，能看到容器中进程的 CPU 占用只有默认 CPU 占用的一半。

```
%Cpu(s): 99.7 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
KiB Mem :  2048416 total,  1284324 free,   479940 used,   284152 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  1407116 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 3290 bingshe+  20   0  212284 205000    212 R 66.4 10.0   0:04.49 stress
 3273 root      20   0  212284 205004    212 R 33.2 10.0   0:16.01 stress
```

3.3　增加管道及环境变量识别

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-3.3
```

本节会给 3.1.2 节已经实现的 run 命令增加一些功能，添加管道和环境变量的识别功能。

当在 Linux 上创建两个进程时，进程之间的通信一般就会使用管道的机制。所谓管道，就是一个连接两个进程的通道，它是 Linux 支持 IPC 的其中一种方式。一般来说，管道都是半双工的，一端进行写操作，另外一端进行读操作。

常用的管道分为两种类型。一种类型是无名管道，它一般用于具有亲缘关系的进程之间；

另外一种是有名管道，或者叫 FIFO 管道，它是一种存在于文件系统的管道，可以被两个没有任何亲缘关系的进程进行访问。有名管道一般可以通过 mkfifo() 函数来创建。

从本质上来说，管道也是文件的一种，但是它和文件通信的区别在于，管道有一个固定大小的缓冲区，大小一般是 4KB。当管道被写满时，写进程就会被阻塞，直到有读进程把管道的内容读出来。同样地，当读进程从管道内拿数据的时候，如果这时管道的内容是空的，那么读进程同样会被阻塞，一直等到有写进程向管道内写数据。

在 3.2 节中实现的那个简单版本的 run 命令有一个缺陷，就是传递参数。在父进程和子进程之间传参，是通过调用命令后面跟上参数，也就是 /proc/self/exe init args 这种方式进行的，然后，在 init 进程内去解析这个参数，执行相应的命令。但是，这有一个缺点就是，如果用户输入的参数很长，或者其中带有一些特殊字符，那么这种方案就会失败了。其实，runC 实现的方案是通过匿名管道来实现父子进程之间通信的，下面就会修改上一版本的代码，增加这个功能。

```
func NewPipe() (*os.File, *os.File, error) {
    read, write, err := os.Pipe()
    if err != nil {
        return nil, nil, err
    }
    return read, write, nil
}
```

首先增加了一个函数，使用 Go 提供的 pipe 方法生成一个匿名管道。这个函数返回两个变量，一个是读一个是写，其类型都是文件类型。

```
func NewParentProcess(tty bool) (*exec.Cmd, *os.File) {
    readPipe, writePipe, err := NewPipe()
    if err != nil {
        log.Errorf("New pipe error %v", err)
        return nil, nil
    }
    cmd := exec.Command("/proc/self/exe", "init")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_
        NEWNS |
            syscall.CLONE_NEWNET | syscall.CLONE_NEWIPC,
    }
    if tty {
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
    }
    // 注意，改动在这里，在这个地方传入管道文件读取端的句柄
```

```
    cmd.ExtraFiles = []*os.File{readPipe}
    return cmd, writePipe
}
```

这个方法的改动点已经表明，这里主要解决的问题是如何将创建的管道的一端传给子进程。这是一个文件类型，肯定不能通过字符参数的方式进行传递，因此使用了 command 的 cmd.ExtraFiles 方法。这个属性的意思是会外带着这个文件句柄去创建子进程。为什么叫“外带着”呢？因为 1 个进程默认会有 3 个文件描述符，分别是标准输入、标准输出、标准错误。这 3 个是子进程一创建的时候就会默认带着的，那么外带的这个文件描述符理所当然地就成为了第 4 个。

```
[vagrant] ll /proc/self/fd
total 0
lrwx------ 1 root root 64 Nov 29 11:45 0 -> /dev/pts/5
lrwx------ 1 root root 64 Nov 29 11:45 1 -> /dev/pts/5
lrwx------ 1 root root 64 Nov 29 11:45 2 -> /dev/pts/5
lr-x------ 1 root root 64 Nov 29 11:45 3 -> /proc/20765/fd
```

这里可以看到默认的文件描述符。通过这种方式，就把管道的一端传给子进程了。下面来看子进程的改动。

```
func readUserCommand() []string {
    //uintptr(3) 就是指 index 为 3 的文件描述符，也就是传递进来的管道的一端
    pipe := os.NewFile(uintptr(3), "pipe")
    msg, err := ioutil.ReadAll(pipe)
    if err != nil {
        log.Errorf("init read pipe error %v", err)
        return nil
    }
    msgStr := string(msg)
    return strings.Split(msgStr, " ")
}
```

这里得到父进程传递过来的管道一端后，就直接去读，由于此时父进程可能还没写入，所以此时的读操作就会停在这里等待输入。然后，看一下最终的 init 方法做了什么。

```
func RunContainerInitProcess() error {
    cmdArray := readUserCommand()
    if cmdArray == nil || len(cmdArray) == 0 {
       return fmt.Errorf("Run container get user command error, cmdArray is nil")
    }
```

```
    defaultMountFlags := syscall.MS_NOEXEC | syscall.MS_NOSUID | syscall.MS_NODEV
    syscall.Mount("proc", "/proc", "proc", uintptr(defaultMountFlags), "")
    // 改动，调用 exec.LookPath，可以在系统的 PATH 里面寻找命令的绝对路径
    path, err := exec.LookPath(cmdArray[0])
    if err != nil {
        log.Errorf("Exec loop path error %v", err)
        return err
    }
    log.Infof("Find path %s", path)
    if err := syscall.Exec(path, cmdArray[0:], os.Environ()); err != nil {
        log.Errorf(err.Error())
    }
    return nil
}
```

可以看到，init 进程读取了父进程传递过来的参数后，在子进程内进行了执行，这样就完成了将用户指定命令传递给子进程的操作。

另外，不知道大家有没有发现，上一版本的代码里面，在执行命令的时候，必须将命令写完全，比如 ls 必须写成 /bin/ls，这是因为 SYS_EXECVE 系统调用并不会去帮我们在 PATH 里面搜寻命令，所以我们多加了一个操作：exec.LookPath(cmdArray[0])。这个函数帮我们在当前系统的 PATH 里面去寻找命令的绝对路径，然后运行起来。

最后，来看一下父进程是在哪里发送的参数。

```
func Run(tty bool, comArray []string) {
    parent, writePipe := container.NewParentProcess(tty)
    if parent == nil {
        log.Errorf("New parent process error")
        return
    }
    if err := parent.Start(); err != nil {
        log.Error(err)
    }
    //发送用户命令
    sendInitCommand(comArray, writePipe)
    parent.Wait()
    os.Exit(0)
}

func sendInitCommand(comArray []string, writePipe *os.File) {
    command := strings.Join(comArray, " ")
    log.Infof("command all is %s", command)
    writePipe.WriteString(command)
```

```
    writePipe.Close()
}
```

在父进程中，当进程 start 完毕后才发送参数，也就是说，子进程这时应该已经在读等待了。流程图如图 3.3 所示。

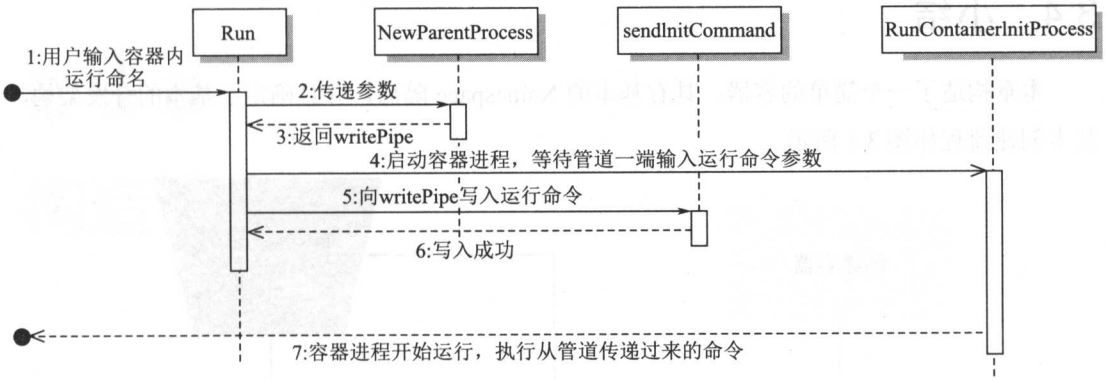


图 3.3

最后，演示一下效果。

```
→  [mydocker] ./mydocker run -ti ls -l
{"level":"info","msg":"command all is ls -l","time":"2016-11-29T11:59:58Z"}
{"level":"info","msg":"init come on","time":"2016-11-29T11:59:58Z"}
{"level":"info","msg":"Find path /bin/ls","time":"2016-11-29T11:59:58Z"}
total 3764
drwxr-xr-x 1 vagrant vagrant     136 Nov 28 15:14 container
drwxr-xr-x 1 vagrant vagrant     136 Nov 26 05:36 Godeps
-rw-r--r-- 1 vagrant vagrant     959 Nov 28 15:14 main_command.go
-rw-r--r-- 1 vagrant vagrant     693 Nov 26 07:53 main.go
-rwxr-xr-x 1 vagrant vagrant 3840478 Nov 29 11:59 mydocker
-rw-r--r-- 1 vagrant vagrant     639 Nov 28 15:08 run.go
drwxr-xr-x 1 vagrant vagrant     136 Nov 26 06:47 vendor
```

可以看到，执行了 ls -l 命令后，程序帮我们找到了 ls 的绝对路径是 /bin/ls，然后，通过管道的方式把命令传递给子进程，子进程进行了执行并返回了结果。

```
→  [mydocker] ./mydocker run -ti bash
{"level":"info","msg":"command all is bash","time":"2016-11-29T12:01:32Z"}
{"level":"info","msg":"init come on","time":"2016-11-29T12:01:32Z"}
{"level":"info","msg":"Find path /bin/bash","time":"2016-11-29T12:01:32Z"}
root@ubuntu:[mydocker]# cd
root@vagrant-ubuntu-trusty-64:~# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
```

```
root          1     0  0 12:01 pts/5    00:00:00 bash
root         27     1  0 12:01 pts/5    00:00:00 ps -ef
```

这里在容器启动时，指定执行了一下 bash，发现容器中已经启动了 bash 的进程。

3.4 小结

本章构造了一个简单的容器，具有基本的 Namespace 隔离，并且确定了基本的开发架构，基本创建流程如图 3.4 所示。

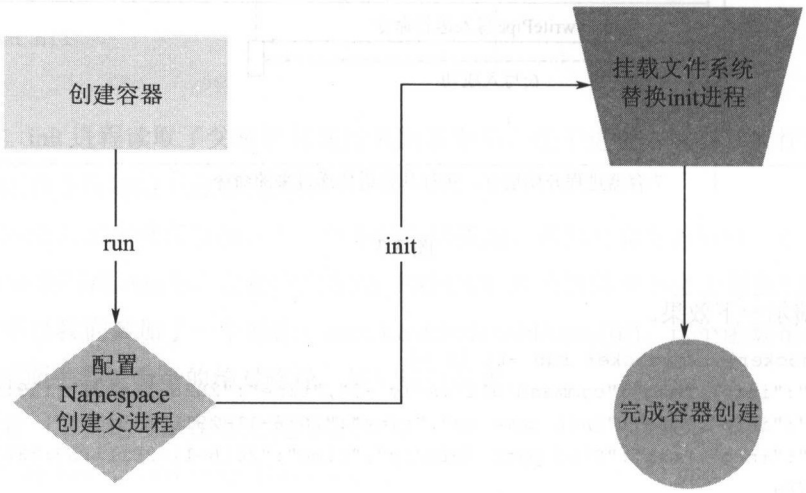


图 3.4

对于 Cgroups， 通过这一章在容器上增加可配置的选项，可以实现对于容器可用资源的控制。最后，使用管道机制将用户输入的命令传递给容器初始化进程，实现了数据的传递。

接下来会继续完善这个容器。

第4章 构造镜像

4.1 使用 busybox 创建容器

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-4.1
```

第 3 章使用了 Namespace 和 Cgroups 技术创建了一个简单的容器，但是大家应该可以发现，容器内的目录还是当前运行程序的目录。而且如果运行一下 mount 命令会发现，可以看到继承自父进程的所有挂载点，这貌似和平常使用的容器表现不同，因为这里缺少了镜像这么一个重要的特性。Docker 镜像可以说是一项伟大的创举，它使得容器传递和迁移更加简单，那么这一节会做一个简单的镜像，让容器跑在有镜像的环境中。

4.1.1 busybox

首先使用一个最精简的镜像——busybox。busybox 是一个集合了非常多 UNIX 工具的箱子，它可以提供非常多在 UNIX 环境下经常使用的命令，可以说 busybox 提供了一个非常完整而且小巧的系统。本小节会先使用它来作为第一个容器内运行的文件系统。

获得 busybox 文件系统的 rootfs 很简单，可以使用 docker export 将一个镜像打成一个 tar 包。首先做如下操作。

```
docker pull busybox
docker run -d busybox top -b
```

```
docker export -o busybox.tar 835360ff16b8(容器ID)
tar -xvf busybox.tar -C busybox/
```

这里使用 Docker 的镜像 busybox，结构如下。

```
→  [busybox] ls -l
total 64K
drwxrwxr-x 2 root root 4.0K Apr 28  2015 bin
drwxr-xr-x 3 root root 4.0K Apr 28  2015 dev
drwxr-xr-x 6 root root 4.0K Apr 28  2015 etc
drwxrwxr-x 3 root root 4.0K Mar  1  2015 home
drwxrwxr-x 2 root root 4.0K Apr 28  2015 lib
lrwxrwxrwx 1 root root    3 Apr 28  2015 lib64 -> lib
lrwxrwxrwx 1 root root   11 Apr 28  2015 linuxrc -> bin/busybox
drwxrwxr-x 2 root root 4.0K Mar  1  2015 media
drwxrwxr-x 2 root root 4.0K Mar  1  2015 mnt
drwxrwxr-x 2 root root 4.0K Mar  1  2015 opt
drwxrwxr-x 2 root root 4.0K Mar  1  2015 proc
drwx------ 2 root root 4.0K Nov 30 11:41 root
drwxrwxr-x 2 root root 4.0K Mar  1  2015 run
drwxr-xr-x 2 root root 4.0K Apr 28  2015 sbin
drwxrwxr-x 2 root root 4.0K Mar  1  2015 sys
drwxrwxrwt 3 root root 4.0K Apr 28  2015 tmp
drwxrwxr-x 6 root root 4.0K Apr 28  2015 usr
drwxrwxr-x 4 root root 4.0K Apr 28  2015 var
```

4.1.2 pivot_root

pivot_root 是一个系统调用，主要功能是去改变当前的 root 文件系统。pivot_root 可以将当前进程的 root 文件系统移动到 put_old 文件夹中，然后使 new_root 成为新的 root 文件系统。new_root 和 put_old 必须不能同时存在当前 root 的同一个文件系统中。pivot_root 和 chroot 的主要区别是，pivot_root 是把整个系统切换到一个新的 root 目录，而移除对之前 root 文件系统的依赖，这样你就能够 umount 原先的 root 文件系统。而 chroot 是针对某个进程，系统的其他部分依旧运行于老的 root 目录中。下面，把代码来实现一下。

```
func pivotRoot(root string) error {
    /*
    为了使当前 root 的老 root 和新 root 不在同一个文件系统下，我们把 root 重新 mount 了一次，
    bind mount 是把相同的内容换了一个挂载点的挂载方法。
    */
    if err := syscall.Mount(root, root, "bind", syscall.MS_BIND|syscall.MS_REC,
    ""); err != nil {
        return fmt.Errorf("Mount rootfs to itself error: %v", err)
```

```
    }
    // 创建 rootfs/.pivot_root 存储 old_root
    pivotDir := filepath.Join(root, ".pivot_root")
    if err := os.Mkdir(pivotDir, 0777); err != nil {
        return err
    }
    // pivot_root 到新的 rootfs, 老的 old_root 现在挂载在 rootfs/.pivot_root 上
    // 挂载点目前依然可以在 mount 命令中看到
    if err := syscall.PivotRoot(root, pivotDir); err != nil {
        return fmt.Errorf("pivot_root %v", err)
    }
    // 修改当前的工作目录到根目录
    if err := syscall.Chdir("/"); err != nil {
        return fmt.Errorf("chdir / %v", err)
    }

    pivotDir = filepath.Join("/", ".pivot_root")
    // umount rootfs/.pivot_root
    if err := syscall.Unmount(pivotDir, syscall.MNT_DETACH); err != nil {
        return fmt.Errorf("unmount pivot_root dir %v", err)
    }
    // 删除临时文件夹
    return os.Remove(pivotDir)
}
```

有了这个函数后，就可以在 init 容器进程的时候，进行一系列的 mount 操作。

```
/*
init 挂载点
*/
func setUpMount() {
    // 获取当前路径
    pwd, err := os.Getwd()
    if err != nil {
        log.Errorf("Get current location error %v", err)
        return
    }
    log.Infof("Current location is %s", pwd)
    pivotRoot(pwd)

    //mount proc
    defaultMountFlags := syscall.MS_NOEXEC | syscall.MS_NOSUID | syscall.MS_NODEV
    syscall.Mount("proc", "/proc", "proc", uintptr(defaultMountFlags), "")

    syscall.Mount("tmpfs", "/dev", "tmpfs", syscall.MS_NOSUID|syscall.MS_
STRICTATIME, "mode=755")
}
```

其中，tmpfs 是一种基于内存的文件系统，可以使用 RAM 或 swap 分区来存储。下面把下载好的 busybox 放到 /root/busybox 宿主机的目录下，使用 cmd.Dir = "/root/busybox" 这个方法给创建出来的子进程指定容器初始化后的工作目录，然后就会运行前面讲到的那些进程，挂载 rootfs 然后把当前目录虚拟成根目录。下面运行一下来看看效果。

```
→  [mydocker] ./mydocker run -ti sh
{"level":"info","msg":"command all is sh","time":"2016-11-30T14:30:04Z"}
{"level":"info","msg":"init come on","time":"2016-11-30T14:30:04Z"}
{"level":"info","msg":"Current location is /root/busybox","time":"2016-11-
30T14:30:04Z"}
{"level":"info","msg":"Find path /bin/sh","time":"2016-11-30T14:30:04Z"}
# 可以看到，容器内的当前目录已经被虚拟定位到了根目录，其实这是在宿主机上映射的 /root/busybox
/ # pwd
/
/ # ls -l
total 56
drwxrwxr-x    2 root     root          4096 Apr 28  2015 bin
drwxr-xr-x    2 root     root            40 Nov 30 14:30 dev
drwxr-xr-x    6 root     root          4096 Apr 28  2015 etc
drwxrwxr-x    3 root     root          4096 Mar  1  2015 home
drwxrwxr-x    2 root     root          4096 Apr 28  2015 lib
lrwxrwxrwx    1 root     root             3 Apr 28  2015 lib64 -> lib
lrwxrwxrwx    1 root     root            11 Apr 28  2015 linuxrc -> bin/busybox
drwxrwxr-x    2 root     root          4096 Mar  1  2015 media
drwxrwxr-x    2 root     root          4096 Mar  1  2015 mnt
drwxrwxr-x    2 root     root          4096 Mar  1  2015 opt
dr-xr-xr-x  114 root     root             0 Nov 30 14:30 proc
drwx------    2 root     root          4096 Nov 30 11:41 root
drwxrwxr-x    2 root     root          4096 Mar  1  2015 run
drwxr-xr-x    2 root     root          4096 Apr 28  2015 sbin
drwxrwxr-x    2 root     root          4096 Mar  1  2015 sys
drwxrwxrwt    3 root     root          4096 Apr 28  2015 tmp
drwxrwxr-x    6 root     root          4096 Apr 28  2015 usr
drwxrwxr-x    4 root     root          4096 Apr 28  2015 var

# 从此处执行 mount 的返回值来看，只有指定的 mount 目录，没有原来从父进程那边继承的设备，说明更换
root 文件系统是成功的。
/ # mount
rootfs on / type rootfs (rw)
/dev/disk/by-uuid/a742ba82-8430-4d30-b747-99c9c9af3168 on / type ext4
(rw,relatime,data=ordered)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,mode=755)
/ #
```

4.2　使用 AUFS 包装 busybox

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-4.2
```

在 4.1 节中，介绍了 UFS 和 AUFS，并通过命令行进行实验得出使用 AUFS 存储 Docker 镜像和容器的大致结构。Docker 在使用镜像启动一个容器时，会新建 2 个 layer：write layer 和 container-init layer。write layer 是容器唯一的可读写层；而 container-init layer 是为容器新建的只读层，用来存储容器启动时传入的系统信息（前面也提到过，在实际的场景下，它们并不是以 write layer 和 container-init layer 命名的）。最后把 write layer、container-init layer 和相关镜像的 layers 都 mount 到一个 mnt 目录下，然后把这个 mnt 目录作为容器启动的根目录。

在 4.1 节中已经实现了使用宿主机 /root/busybox 目录作为文件的根目录，但在容器内对文件的操作仍然会直接影响到宿主机的 /root/busybox 目录。本节要进一步进行容器和镜像隔离，实现在容器中进行的操作不会对镜像产生任何影响的功能。

NewWorkSpace 函数是用来创建容器文件系统的，它包括 CreateReadOnlyLayer、CreateWriteLayer 和 CreateMountPoint。

- CreateReadOnlyLayer 函数新建 busybox 文件夹，将 busybox.tar 解压到 busybox 目录下，作为容器的只读层。
- CreateWriteLayer 函数创建了一个名为 writeLayer 的文件夹，作为容器唯一的可写层。
- 在 CreateMountPoint 函数中，首先创建了 mnt 文件夹，作为挂载点，然后把 writeLayer 目录和 busybox 目录 mount 到 mnt 目录下。

最后，在 NewParentProcess 函数中将容器使用的宿主机目录 /root/busybox 替换成 /root/mnt。

```
func NewWorkSpace(rootURL string, mntURL string) {
    CreateReadOnlyLayer(rootURL)
    CreateWriteLayer(rootURL)
    CreateMountPoint(rootURL, mntURL)
}

// 将 busybox.tar 解压到 busybox 目录下，作为容器的只读层
func CreateReadOnlyLayer(rootURL string) {
    busyboxURL := rootURL + "busybox/"
    busyboxTarURL := rootURL + "busybox.tar"
    exist, err := PathExists(busyboxURL)
```

```
    if err != nil {
        log.Infof("Fail to judge whether dir %s exists. %v", busyboxURL, err)
    }
    if exist == false {
        if err := os.Mkdir(busyboxURL, 0777); err != nil {
            log.Errorf("Mkdir dir %s error. %v", busyboxURL, err)
        }
        if _, err := exec.Command("tar", "-xvf", busyboxTarURL, "-C", busyboxURL).
CombinedOutput(); err != nil {
            log.Errorf("unTar dir %s error %v", busyboxTarURL, err)
        }
    }
}

// 创建了一个名为 writeLayer 的文件夹作为容器唯一的可写层
func CreateWriteLayer(rootURL string) {
    writeURL := rootURL + "writeLayer/"
    if err := os.Mkdir(writeURL, 0777); err != nil {
        log.Errorf("Mkdir dir %s error. %v", writeURL, err)
    }
}


func CreateMountPoint(rootURL string, mntURL string) {
    // 创建 mnt 文件夹作为挂载点
    if err := os.Mkdir(mntURL, 0777); err != nil {
        log.Errorf("Mkdir dir %s error. %v", mntURL, err)
    }
    // 把 writeLayer 目录和 busybox 目录 mount 到 mnt 目录下
    dirs := "dirs=" + rootURL + "writeLayer:" + rootURL + "busybox"
    cmd := exec.Command("mount", "-t", "aufs", "-o", dirs, "none", mntURL)
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    if err := cmd.Run(); err != nil {
        log.Errorf("%v", err)
    }
}

// 判断文件路径是否存在
func PathExists(path string) (bool, error) {
    _, err := os.Stat(path)
    if err == nil {
        return true, nil
    }
    if os.IsNotExist(err) {
```

```
        return false, nil
    }
    return false, err
}
```

接下来，在 NewParentProcess 函数中将容器使用的宿主机目录 /root/busybox 替换成 /root/mnt。这样，使用 AUFS 系统启动容器的代码就完成了。

```
cmd.ExtraFiles = []*os.File{readPipe}
mntURL := "/root/mnt/"
rootURL := "/root/"
NewWorkSpace(rootURL, mntURL)
cmd.Dir = mntURL
return cmd, writePipe
```

Docker 会在删除容器的时候，把容器对应的 Write Layer 和 Container-init Layer 删除，而保留镜像所有的内容。本节中，在容器退出的时候会删除 Write Layer。DeleteWorkSpace 函数包括 DeleteMountPoint 和 DeleteWriteLayer。

- 首先，在 DeleteMountPoint 函数中 umount mnt 目录。
- 然后，删除 mnt 目录。
- 最后，在 DeleteWriteLayer 函数中删除 writeLayer 文件夹。这样容器对文件系统的更改就都已经抹去了。

```
func DeleteWorkSpace(rootURL string, mntURL string){
    DeleteMountPoint(rootURL, mntURL)
    DeleteWriteLayer(rootURL)
}

func DeleteMountPoint(rootURL string, mntURL string){
    cmd := exec.Command("umount", mntURL)
    cmd.Stdout=os.Stdout
    cmd.Stderr=os.Stderr
    if err := cmd.Run(); err != nil {
        log.Errorf("%v",err)
    }
    if err := os.RemoveAll(mntURL); err != nil {
        log.Errorf("Remove dir %s error %v", mntURL, err)
    }
}

func DeleteWriteLayer(rootURL string) {
    writeURL := rootURL + "writeLayer/"
```

```
    if err := os.RemoveAll(writeURL); err != nil {
        log.Errorf("Remove dir %s error %v", writeURL, err)
    }
}
```

流程图如图 4.1 所示。

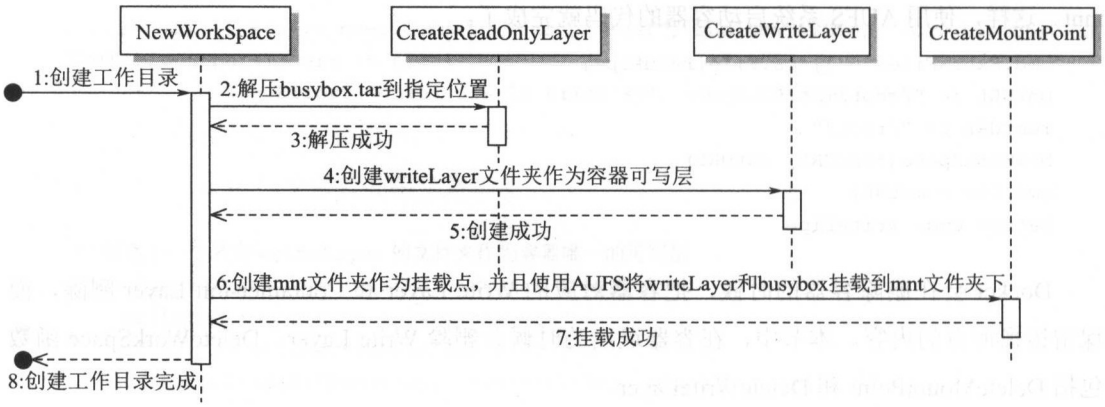


图 4.1

下面，运行 mydocker 来测试一下代码。首先查看一下 /root 目录下的内容，发现只有 busybox.tar。

```
ls /root
busybox.tar
```

启动一个容器。

```
# ./mydocker run -ti sh
{"level":"info","msg":"command all is sh","time":"2016-12-18T23:09:36+08:00"}
{"level":"info","msg":"init come on","time":"2016-12-18T23:09:36+08:00"}
{"level":"info","msg":"Current location is /root/mnt","time":"2016-12-
18T23:09:36+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2016-12-18T23:09:36+08:00"}
```

再次查看 /root 目录下的内容，可以看到 root 目录下多了 busybox、writeLayer 和 mnt 三个目录。

```
# ls /root
busybox  busybox.tar  mnt  writeLayer
```

在容器中新建一个文件夹。

```
/ # ls
```

```
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ # mkdir tmp/qinyujia.txt
/ # ls tmp
qinyujia.txt
```

新建一个 cmd 窗口，分别查看 busybox 和 writeLayer 目录的内容。可以看到 busybox 目录的内容并没有改变，而 writeLayer 文件夹中多了一个 tmp 目录，tmp 目录中存放着刚刚创建的 qinyujia.txt 文件。至此，说明容器并未直接对镜像层中的 tmp 目录进行操作，而是在 writeLayer 层复制 tmp 目录，修改其内容。

```
root@iZ62wrfki2jZ:~# ls /root/busybox
bin  dev  etc  home  proc  root  sys  tmp  usr  var
root@iZ62wrfki2jZ:~# ls /root/writeLayer
root  tmp
root@iZ62wrfki2jZ:~# ls /root/writeLayer/tmp
qinyujia.txt
```

在容器中执行 exit 退出容器，然后再次查看宿主机上的 root 文件夹内容。writeLayer 和 mnt 目录被删除，作为镜像的 busybox 层仍然保留，并且内容未被修改。

```
# ls /root
busybox  busybox.tar
```

4.3　实现 volume 数据卷

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-4.3
```

上一小节介绍了如何使用 AUFS 包装 busybox，从而实现容器和镜像的分离。但是一旦容器退出，容器可写层的所有内容都会被删除。那么，如果用户需要持久化容器里的部分数据该怎么办呢？ volume 就是用来解决这个问题的。本节将会介绍如何实现将宿主机的目录作为数据卷挂载到容器中，并且在容器退出后，数据卷中的内容仍然能够保存在宿主机上。

使用 AUFS 创建容器文件系统的实现过程如下。

启动容器的时候：

1. 创建只读层（busybox）；

2. 创建容器读写层（writeLayer）；

3. 创建挂载点（mnt），并把只读层和读写层挂载到挂载点；

4. 将挂载点作为容器的根目录。

容器退出的时候：

1. 卸载挂载点（mnt）的文件系统；

2. 删除挂载点；

3. 删除读写层（writeLayer）。

本节要在这个基础上添加绑定宿主机文件夹到容器数据卷的功能。首先，在 main_command.go 文件的 runCommand 命令中添加 -v 标签。

```
var runCommand = cli.Command{
    Name: "run",
    Usage: `Create a container with namespace and cgroups limit
           mydocker run -ti [command]`,
    Flags: []cli.Flag{
        cli.BoolFlag{
            Name:  "ti",
            Usage: "enable tty",
        },
        //添加 -v 标签
        cli.StringFlag{
            Name:  "v",
            Usage: "volume",
        },
    },
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Missing container command")
        }
        var cmdArray []string
        for _, arg := range context.Args() {
            cmdArray = append(cmdArray, arg)
        }
        tty := context.Bool("ti")
        //把 volume 参数传给 Run 函数
        volume := context.String("v")
        Run(tty, cmdArray, volume)
        return nil
    },
}
```

在 Run 函数中，把 volume 传给创建容器的 NewParentProcess 函数和删除容器文件系统的 DeleteWorkSpace 函数。

```
func Run(tty bool, comArray []string, volume string) {
    //修改点
    parent, writePipe := container.NewParentProcess(tty, volume)
    if parent == nil {
        log.Errorf("New parent process error")
        return
    }
    if err := parent.Start(); err != nil {
        log.Error(err)
    }
    sendInitCommand(comArray, writePipe)
    parent.Wait()
    mntURL := "/root/mnt"
    rootURL := "/root"
    //修改点
    container.DeleteWorkSpace(rootURL, mntURL, volume)
    os.Exit(0)
}
```

在 NewWorkSpace 函数中，继续把 volume 值传给创建容器文件系统的 NewWorkSpace 方法。

```
NewWorkSpace(rootURL, mntURL, volume)
```

创建容器文件系统的过程如下。

1. 创建只读层。
2. 创建容器读写层。
3. 创建挂载点并把只读层和读写层挂载到挂载点上。
4. 接下来，首先判断 volume 是否为空，如果是，就表示用户并没有使用挂载标签，结束创建过程。
5. 如果不为空，则使用 volumeUrlExtract 函数解析 volume 字符串。
6. 当 volumeUrlExtract 函数返回的字符数组长度为 2，并且数据元素均不为空的时候，则执行 MountVolume 函数来挂载数据卷。
7. 否则，提示用户创建数据卷输入值不对。

```
func NewWorkSpace(rootURL string, mntURL string, volume string) {
    CreateReadOnlyLayer(rootURL)
    CreateWriteLayer(rootURL)
    CreateMountPoint(rootURL, mntURL)
    //根据 volume 判断是否执行挂载数据卷操作
    if volume != "" {
        volumeURLs := volumeUrlExtract(volume)
```

```
        length := len(volumeURLs)
        if length == 2 && volumeURLs[0] != "" && volumeURLs[1] != "" {
            MountVolume(rootURL, mntURL, volumeURLs)
            log.Infof("%q",volumeURLs)
        }else{
            log.Infof("Volume parameter input is not correct.")
        }
    }
}
// 解析 volume 字符串
func volumeUrlExtract(volume string) ([]string) {
    var volumeURLs []string
    volumeURLs =  strings.Split(volume, ":")
    return volumeURLs
}
```

挂载数据卷的过程如下。

1. 首先，读取宿主机文件目录 URL，创建宿主机文件目录（/root/${parentUrl}）。
2. 然后，读取容器挂载点 URL，在容器文件系统里创建挂载点（/root/mnt/${containerUrl}）。
3. 最后，把宿主机文件目录挂载到容器挂载点。这样启动容器的过程，对数据卷的处理也就完成了。

```
func MountVolume(rootURL string, mntURL string, volumeURLs []string)  {
    // 创建宿主机文件目录
    parentUrl := volumeURLs[0]
    if err := os.Mkdir(parentUrl, 0777); err != nil {
        log.Infof("Mkdir parent dir %s error. %v", parentUrl, err)
    }
    // 在容器文件系统里创建挂载点
    containerUrl := volumeURLs[1]
    containerVolumeURL := mntURL + containerUrl
    if err := os.Mkdir(containerVolumeURL, 0777); err != nil {
        log.Infof("Mkdir container dir %s error. %v", containerVolumeURL, err)
    }
    // 把宿主机文件目录挂载到容器挂载点
    dirs := "dirs=" + parentUrl
    cmd := exec.Command("mount", "-t", "aufs", "-o", dirs, "none", containerVolumeURL)
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    if err := cmd.Run(); err != nil {
        log.Errorf("Mount volume failed. %v", err)
    }
}
```

删除容器文件系统的过程如下。

1. 只有在 volume 不为空，并且使用 volumeUrlExtract 函数解析 volume 字符串返回的字符数组长度为 2，数据元素均不为空的时候，才执行 DeleteMountPointWithVolume 函数来处理。

2. 其余的情况仍然使用前面的 DeleteMountPoint 函数。

```
func DeleteWorkSpace(rootURL string, mntURL string, volume string){
    if(volume != ""){
        volumeURLs := volumeUrlExtract(volume)
        length := len(volumeURLs)
        if(length == 2 && volumeURLs[0] != "" && volumeURLs[1] !=""){
            DeleteMountPointWithVolume(rootURL, mntURL, volumeURLs)
        }else{
            DeleteMountPoint(rootURL, mntURL)
        }
    }else {
        DeleteMountPoint(rootURL, mntURL)
    }
    DeleteWriteLayer(rootURL)
}
```

DeleteMountPointWithVolume 函数的处理逻辑如下。

1. 首先，卸载 volume 挂载点的文件系统（/root/mnt/${containerUrl}），保证整个容器的挂载点没有被使用。

2. 然后，再卸载整个容器文件系统的挂载点（/root/mnt）。

3. 最后，删除容器文件系统挂载点。整个容器退出过程中的文件系统处理就结束了。

```
func DeleteMountPointWithVolume(rootURL string, mntURL string, volumeURLs []
string){
    // 卸载容器里 volume 挂载点的文件系统
    containerUrl := mntURL + volumeURLs[1]
    cmd := exec.Command("umount", containerUrl)
    cmd.Stdout=os.Stdout
    cmd.Stderr=os.Stderr
    if err := cmd.Run(); err != nil {
        log.Errorf("Umount volume failed. %v",err)
    }
    // 卸载整个容器文件系统的挂载点
    cmd = exec.Command("umount", mntURL)
    cmd.Stdout=os.Stdout
    cmd.Stderr=os.Stderr
    if err := cmd.Run(); err != nil {
        log.Errorf("Umount mountpoint failed. %v",err)
```

```
    }
    // 删除容器文件系统挂载点
    if err := os.RemoveAll(mntURL); err != nil {
        log.Infof("Remove mountpoint dir %s error %v", mntURL, err)
    }
}
```

流程图如图 4.2 所示。

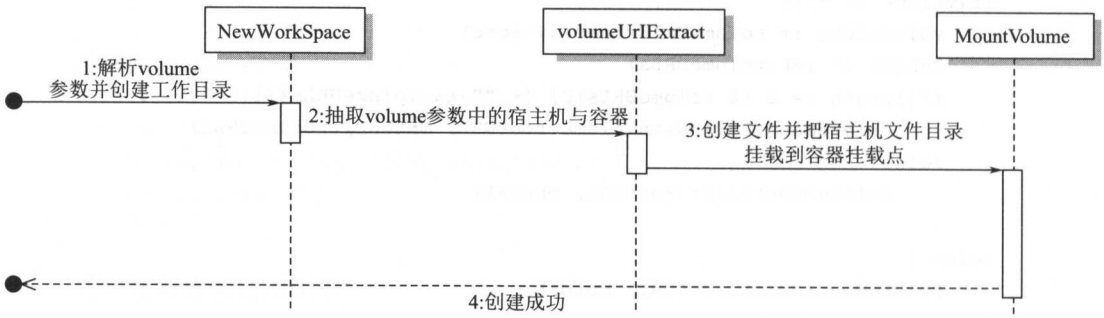


图 4.2

下面来验证一下程序的正确性。第一个实验是把一个宿主机上不存在的文件目录挂载到容器中。首先查看 root 目录下的文件，只有 busybox.tar。

```
# ls /root
busybox.tar
```

启动一个容器，把宿主机的 /root/volume 挂载到容器的 /containerVolume 目录下。

```
# ./mydocker run -ti -v /root/volume:/containerVolume sh
{"level":"info","msg":"[\"/root/volume\" \"/containerVolume\"]","time":"2016-12-25T18:43:26+08:00"}
{"level":"info","msg":"command all is sh","time":"2016-12-25T18:43:26+08:00"}
{"level":"info","msg":"init come on","time":"2016-12-25T18:43:26+08:00"}
{"level":"info","msg":"Current location is /root/mnt","time":"2016-12-25T18:43:26+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2016-12-25T18:43:26+08:00"}
/ #
```

另外打开一个 terminal 窗口，查看宿主机 /root 目录的内容，多了 mnt、writeLayer 和 volume 三个目录。

```
# ls /root
busybox  busybox.tar  mnt  volume  writeLayer
```

在容器里执行 ls 命令，多了一个 containerVolume 目录。

```
/ # ls
```

```
bin              dev          home          root          tmp           var
containerVolume  etc          proc          sys           usr
```

进入 containerVolume 目录，创建一个 test.txt 文件，写下 hello world。

```
/ # cd containerVolume/
/containerVolume # echo -e "hello world" >> /containerVolume/test.txt
/containerVolume # ls
test.txt
/containerVolume # cat test.txt
hello world
```

在另外一个 terminal 窗口，查看宿主机 /root/volume 目录的内容。多了一个写着“hello world”的 test.txt 文件。

```
# ls /root/volume
test.txt
root@iZ62wrfki2jZ:~# cat /root/volume/test.txt
hello world
```

在容器里执行 exit 命令退出。然后查看宿主机 /root 目录的内容。可以看到 volume 文件夹并没有被删除，内容也保持不变。

```
# ls /root
busybox  busybox.tar  volume
root@iZ62wrfki2jZ:~# ls /root/volume/
test.txt
root@iZ62wrfki2jZ:~# cat /root/volume/test.txt
hello world
```

第二个实验是把宿主机上存在并包含文件的文件目录挂载到容器中。这样可以复用上面实验创建的这个 /root/volume 目录。首先，还是查看宿主机 /root 目录的内容。

```
# ls /root
busybox  busybox.tar  volume
root@iZ62wrfki2jZ:~# ls /root/volume/
test.txt
root@iZ62wrfki2jZ:~# cat /root/volume/test.txt
hello world
```

启动一个容器，把宿主机的 /root/volume 挂载到容器的 /containerVolume 目录下。

```
 ./mydocker run -ti -v /root/volume:/containerVolume sh
{"level":"info","msg":"Mkdir parent dir /root/volume error. mkdir /root/volume:
file exists","time":"2017-01-07T15:37:25+08:00"}
{"level":"info","msg":"[\"/root/volume\" \"/containerVolume\"]","time":"2017-01-
```

```
07T15:37:25+08:00"}
{"level":"info","msg":"command all is sh","time":"2017-01-07T15:37:25+08:00"}
{"level":"info","msg":"init come on","time":"2017-01-07T15:37:25+08:00"}
{"level":"info","msg":"Current location is /root/mnt","time":"2017-01-
07T15:37:25+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-07T15:37:25+08:00"}
/ #
```

查看容器文件系统根目录的内容，发现存在 containerVolume，继续查看文件夹 containerVolume 的内容，有一个写着“hello world”的 test.txt 文件。

```
/ # ls
bin              dev              home             root             tmp              var
containerVolume  etc              proc             sys              usr
/ # ls containerVolume/
test.txt
/ # cat /containerVolume/test.txt
hello world
```

进入 containerVolume 目录，在 test.txt 中添加“hello world, again”。

```
/ # cd containerVolume/
/containerVolume # ls
test.txt
/containerVolume # echo -e "hello world, again" >> /containerVolume/test.txt
/containerVolume # cat /containerVolume/test.txt
hello world
hello world, again
```

在 containerVolume 目录下创建一个新的 test-again.txt 文件，在 test-again.txt 中写入“hello world, again”。

```
/containerVolume # echo -e "hello world, again" >> /containerVolume/test-again.txt
/containerVolume # ls
test-again.txt  test.txt
```

另外打开一个 terminal 窗口，查看宿主机 /root/volume 目录的内容，变化与容器 containerVolume 目录一致。

```
root@iZ62wrfki2jZ:~# cd volume/
root@iZ62wrfki2jZ:~/volume# ls
test-again.txt  test.txt
root@iZ62wrfki2jZ:~/volume# cat test.txt
hello world
hello world, again
```

```
root@iZ62wrfki2jZ:~/volume# cat test-again.txt
hello world, again
```

在容器里执行 exit 命令退出。然后查看宿主机 /root 目录的内容。可以看到 volume 文件夹并没有被删除，内容也保持不变。

```
root@iZ62wrfki2jZ:~/volume# ls /root/volume/
test-again.txt  test.txt
root@iZ62wrfki2jZ:~/volume# cat /root/volume/test.txt
hello world
hello world, again
root@iZ62wrfki2jZ:~/volume# cat /root/volume/test-again.txt
hello world, again
```

4.4　实现简单镜像打包

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-4.4
```

容器在退出时会删除所有可写层的内容，mydocker commit 命令的目的就是把运行状态容器的内容存储成镜像保存下来。

首先，在 main.go 文件中添加 commit 命令。

```
app.Commands = []cli.Command{
    initCommand,
    runCommand,
    commitCommand,
}
```

接着，在 main_command.go 文件中实现 commitCommand 命令。从用户的输入获取 image name。

```
var commitCommand = cli.Command{
    Name:  "commit",
    Usage: "commit a container into image",
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Missing container name")
        }
        imageName := context.Args().Get(0)
        //commitContainer(containerName)
        commitContainer(imageName)
        return nil
```

```
    },
}
```

添加 commit.go 文件，通过 commitContainer 函数实现将容器文件系统打包成 ${imagename}.tar 文件。

```
package main

import (
    log "github.com/Sirupsen/logrus"
    "fmt"
    "os/exec"
)

func commitContainer(imageName string){
    mntURL := "/root/mnt"
    imageTar := "/root/" + imageName + ".tar"
    fmt.Printf("%s",imageTar)
     if _, err := exec.Command("tar", "-czf", imageTar, "-C", mntURL, ".").
CombinedOutput(); err != nil {
        log.Errorf("Tar folder %s error %v", mntURL, err)
    }
}
```

流程图如图 4.3 所示。

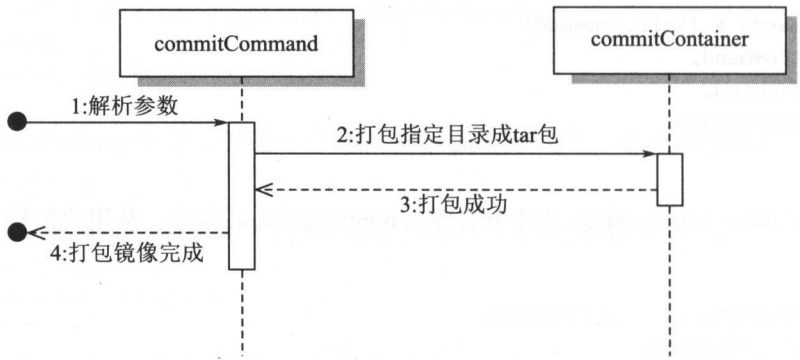


图 4.3

接下来，运行程序来验证代码的正确性。首先，启动一个容器。

```
# ./mydocker run -ti sh
{"level":"info","msg":"command all is sh","time":"2016-12-25T23:05:40+08:00"}
{"level":"info","msg":"init come on","time":"2016-12-25T23:05:40+08:00"}
{"level":"info","msg":"Current location is /root/mnt","time":"2016-12-
25T23:05:40+08:00"}
```

```
{"level":"info","msg":"Find path /bin/sh","time":"2016-12-25T23:05:40+08:00"}
/ #
```

另外打开一个 terminal 窗口，先查看 /root 目录的内容，执行 mydocker commit 命令，再次查看 /root 目录的内容，多了 image.tar 文件。

```
# ls /root
busybox  busybox.tar  mnt  writeLayer
# ./mydocker commit image
/root/image.tar
# ls /root
busybox  busybox.tar  image.tar  mnt  writeLayer
```

解压 image.tar 查看内容。

```
# mkdir untar
# cd untar
# tar -xvf /root/image.tar
# ls
bin  dev  etc  home  proc  root  sys  tmp  usr  var
```

4.5　小结

本章首先使用 busybox 作为基础镜像创建了一个容器，理解了什么是 rootfs，以及如何使用 rootfs 来打造容器的基本运行环境。然后，使用 AUFS 来构建了一个拥有二层模式的镜像，对于最上层可写层的修改不会影响到基础层。这里就基本解释了镜像分层存储的原理。之后使用 -v 参数做了一个 volume 挂载的例子，介绍了如何将容器外部的文件系统挂载到容器中，并且让它可以访问。 最后实现了一个简单版本的容器镜像打包。这一章主要针对镜像的存储及文件系统做了基本的原理性介绍，通过这几个例子，可以很好地理解镜像是如何构建的，第 5 章会基于这些基础做更多的扩展。

第5章 构建容器进阶

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.1
```

经过第4章的铺垫，我们又了解了关于镜像的知识，明白了基本的容器镜像的构成。不过，与Docker创建的容器相比，我们还缺少后台运行的容器，也就是detach类型的容器功能，并且不能通过docker ps查看目前处于运行中的容器，也不能通过docker logs查看容器的输出，更不能通过docker exec进入到一个已经创建好了的容器中。从本章开始，会去一一实现这些功能。

5.1 实现容器的后台运行

通过前面章节的讲解，我们实现了一个可以运行交互命令的容器，但是在容器真正使用的时候，希望它能够在后台运行。这样就需要父进程创建完成子进程后，detach掉子进程。在Docker早期版本，所有的容器init进程都是从docker daemon这个进程fork出来的，这也就会导致一个众所周知的问题，如果docker daemon挂掉，那么所有的容器都会宕掉，这给升级docker daemon带来很大的风险。后来，Docker使用了containerd，也就是现在的runC，便可以实现即使daemon挂掉，容器依然健在的功能了，其结构如图5.1所示。

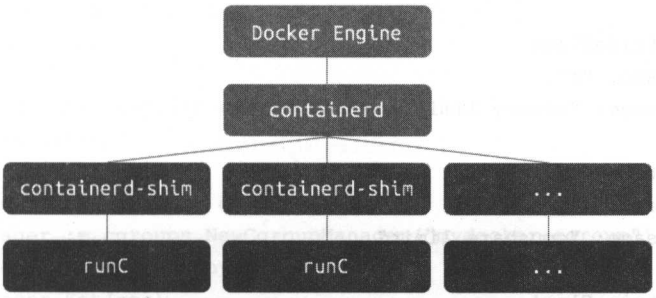


图 5.1

我们并不想去实现一个 daemon，因为这和容器的关联不是特别大，而且，查看 Docker 的运行引擎 runC 可以发现，runC 也提供一种 detach 功能，可以保证在 runC 退出的情况下容器依然可以运行。因此，我们将会使用 detach 功能去实现创建完成容器后，mydocker 就会退出，但是容器依然继续运行的功能。

容器，在操作系统看来，其实就是一个进程。当前运行命令的 mydocker 是主进程，容器是被当前 mydocker 进程 fork 出来的子进程。子进程的结束和父进程的运行是一个异步的过程，即父进程永远不知道子进程到底什么时候结束。如果创建子进程的父进程退出，那么这个子进程就成了没人管的孩子，俗称孤儿进程。为了避免孤儿进程退出时无法释放所占用的资源而僵死，进程号为 1 的进程 init 就会接受这些孤儿进程。

这就是父进程退出而容器进程依然运行的原理。虽然容器刚开始是由当前运行的 mydocker 进程创建的，但是当 mydocker 进程退出后，容器进程就会被进程号为 1 的 init 进程接管，这时容器进程还是运行着的，这样就实现了 mydocker 退出、容器不宕掉的功能。

首先，需要在 main-command.go 里面添加 -d 标签，表示这个容器启动的时候后台在运行。

```
var runCommand = cli.Command{
    Name: "run",
    Usage: `Create a container with namespace and cgroups limit
           mydocker run -ti [command]`,
    Flags: []cli.Flag{
        cli.BoolFlag{
            Name:  "ti",
            Usage: "enable tty",
        },
        // 添加 -d 标签
        cli.BoolFlag{
            Name:  "d",
            Usage: "detach container",
```

```
        },
        cli.StringFlag{
            Name: "m",
            Usage: "memory limit",
        },
        cli.StringFlag{
            Name: "cpushare",
            Usage: "cpushare limit",
        },
        cli.StringFlag{
            Name: "cpuset",
            Usage: "cpuset limit",
        },
    },
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Missing container command")
        }
        var cmdArray []string
        for _, arg := range context.Args() {
            cmdArray = append(cmdArray, arg)
        }

        createTty := context.Bool("ti")
        detach := context.Bool("d")

        // 这里的 createTty 和 detach 不能共存
        if createTty && detach {
            return fmt.Errorf("ti and d paramter can not both provided")
        }
        resConf := &subsystems.ResourceConfig{
            MemoryLimit: context.String("m"),
            CpuSet: context.String("cpuset"),
            CpuShare:context.String("cpushare"),
        }
        log.Infof("createTty %v", createTty)
        Run(createTty, cmdArray, resConf)
        return nil
    },
}
```

这里的 createTty 和 detach 不能共存，只能选择其一，后面会根据这个来做一些操作。

```
func Run(tty bool, comArray []string, res *subsystems.ResourceConfig) {
    parent, writePipe := container.NewParentProcess(tty)
    if parent == nil {
```

```
        log.Errorf("New parent process error")
        return
    }
    if err := parent.Start(); err != nil {
        log.Error(err)
    }
    // use mydocker-cgroup as cgroup name
    cgroupManager := cgroups.NewCgroupManager("mydocker-cgroup")
    defer cgroupManager.Destroy()
    cgroupManager.Set(res)
    cgroupManager.Apply(parent.Process.Pid)

    sendInitCommand(comArray, writePipe)
    if tty {
        parent.Wait()
    }
}
```

此处添加了判断，原来 parent.Wait() 主要是用于父进程等待子进程结束，这在交互式创建容器的步骤里面是没问题的，但是在这里，如果 detach 创建了容器，就不能再去等待，创建容器之后，父进程就已经退出了。因此，这里只是将容器内的 init 进程启动起来，就已经完成工作，紧接着就可以退出，然后由操作系统进程 ID 为 1 的 init 进程去接管容器进程。

下面来实验一下。

```
mydocker run -d top
{"level":"info","msg":"createTty false","time":"2016-12-04T09:55:02Z"}
{"level":"info","msg":"command all is top","time":"2016-12-04T09:55:03Z"}
```

使用 top 作为容器内前台进程。然后在宿主机上执行 ps -ef 看一下创建的容器进程是否存在。

```
ps -ef
root     19304 19303  0 09:47 pts/5    00:00:00 zsh
root     19604     1  0 09:55 pts/5    00:00:00 top
root     19612 19304  0 09:57 pts/5    00:00:00 ps -ef
```

可以看到，top 命令的进程正在运行着，它的父进程 ID 很闪亮的是 1，这说明虽然 mydocker 主进程退出了，但是容器进程依然存在，由于父进程消失，它就被进程 ID 为 1 的 init 进程给托管了，由此就实现了 mydocker run -d 命令，即容器的后台运行。在下面的章节中会继续增加其他命令。

5.2 实现查看运行中容器

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.2
```

4.1 节已经实现了 mydocker run -d 命令，可以让容器脱离父进程在后台独立运行，那么我们怎么知道有哪些容器在运行，而且它们的信息又是什么呢？这里就需要实现 mydocker ps 命令了。其实 mydocker ps 命令比较简单，主要是去约定好的位置查询一下容器的信息数据，然后显示出来，因此数据准备就显得尤为重要。

5.2.1 准备数据

在前面章节创建的容器中，所有关于容器的信息，比如 PID 、容器创建时间、容器运行命令等，都没有记录，这导致容器运行完后就再也不知道它的信息了，因此需要把这部分信息保留下来。首先，要在 main 函数的 run 参数里面增加一个 name 标签，方便用户指定容器的名字。

```
var runCommand = cli.Command{
    Name: "run",
    Usage: `Create a container with namespace and cgroups limit ie: mydocker run
-ti [command]`,
    Flags: []cli.Flag{
        cli.BoolFlag{
            Name:  "ti",
            Usage: "enable tty",
        },
        cli.BoolFlag{
            Name:  "d",
            Usage: "detach container",
        },
        cli.StringFlag{
            Name:  "m",
            Usage: "memory limit",
        },
        cli.StringFlag{
            Name:  "cpushare",
            Usage: "cpushare limit",
        },
        cli.StringFlag{
            Name:  "cpuset",
            Usage: "cpuset limit",
```

```
        },
        // 提供 run 后面的 -name 指定容器名字参数
        cli.StringFlag{
            Name:  "name",
            Usage: "container name",
        },
    },
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Missing container command")
        }
        var cmdArray []string
        for _, arg := range context.Args() {
            cmdArray = append(cmdArray, arg)
        }
        createTty := context.Bool("ti")
        detach := context.Bool("d")

        if createTty && detach {
            return fmt.Errorf("ti and d paramter can not both provided")
        }
        resConf := &subsystems.ResourceConfig{
            MemoryLimit: context.String("m"),
            CpuSet:      context.String("cpuset"),
            CpuShare:    context.String("cpushare"),
        }
        log.Infof("createTty %v", createTty)
        // 将取到的容器名称传递下去，如果没有则取到的值为空
        containerName := context.String("name")
        Run(createTty, cmdArray, resConf, containerName)
        return nil
    },
}
```

然后，需要增加一个 record 方法记录容器的相关信息。在增加之前，需要一个 ID 生成器，用来唯一标识容器。使用过 Docker 的都知道，每个容器都会有一个 ID，为了方便起见，就用 10 位数字来表示一个容器的 ID。

```
func randStringBytes(n int) string {
    letterBytes := "1234567890"
    rand.Seed(time.Now().UnixNano())
    b := make([]byte, n)
    for i := range b {
        b[i] = letterBytes[rand.Intn(len(letterBytes))]
```

```
    }
    return string(b)
}
```

这里以时间戳为种子，每次生成一个 10 以内的数字作为 letterBytes 数组的下标，最后拼接生成整个容器的 ID。另外就是记录容器信息这个重要的环节，我们先定义了一个容器的一些基本信息，比如 PID 和创建时间等，然后默认把容器的信息以 json 的形式存储在宿主机的“/var/run/mydocker/ 容器名 /config.json”文件里面。容器的基本格式如下。

```
type ContainerInfo struct {
    Pid         string `json:"pid"`         // 容器的 init 进程在宿主机上的 PID
    Id          string `json:"id"`          // 容器 ID
    Name        string `json:"name"`        // 容器名
    Command     string `json:"command"`     // 容器内 init 进程的运行命令
    CreatedTime string `json:"createTime"` // 创建时间
    Status      string `json:"status"`      // 容器的状态
}
```

然后定义了如下几个常量。

```
var (
    RUNNING             string = "running"
    STOP                string = "stopped"
    Exit                string = "exited"
    DefaultInfoLocation string = "/var/run/mydocker/%s/"
    ConfigName           string = "config.json"
)
```

下面就可以去记录容器的信息了，如下是记录容器信息的 record 函数的实现。

```
func recordContainerInfo(containerPID int, commandArray []string, containerName
string) (string, error) {
    // 首先生成 10 位数字的容器 ID
    id := randStringBytes(10)
    // 以当前时间为容器创建时间
    createTime := time.Now().Format("2006-01-02 15:04:05")
    command := strings.Join(commandArray, "")
    // 如果用户不指定容器名，那么就以容器 id 当作容器名
    if containerName == "" {
        containerName = id
    }
    // 生成容器信息的结构体实例
    containerInfo := &container.ContainerInfo{
        Id:          id,
```

```
        Pid:         strconv.Itoa(containerPID),
        Command:     command,
        CreatedTime: createTime,
        Status:      container.RUNNING,
        Name:        containerName,
    }

    //将容器信息的对象 json 序列化成字符串
    jsonBytes, err := json.Marshal(containerInfo)
    if err != nil {
        log.Errorf("Record container info error %v", err)
        return "", err
    }
    jsonStr := string(jsonBytes)

    //拼凑一下存储容器信息的路径
    dirUrl := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    //如果该路径不存在，就级联地全部创建
    if err := os.MkdirAll(dirUrl, 0622); err != nil {
        log.Errorf("Mkdir error %s error %v", dirUrl, err)
        return "", err
    }
    fileName := dirUrl + "/" + container.ConfigName
    //创建最终的配置文件——config.json 文件
    file, err := os.Create(fileName)
    defer file.Close()
    if err != nil {
        log.Errorf("Create file %s error %v", fileName, err)
        return "", err
    }
    //将 json 化之后的数据写入到文件中
    if _, err := file.WriteString(jsonStr); err != nil {
        log.Errorf("File write string error %v", err)
        return "", err
    }

    return containerName, nil
}
```

经过上面的步骤，就实现了把创建容器的信息持久化到磁盘的“/var/run/ 容器名 /config.json”文件上。最后，在 Run 主函数上加上对于这个函数的调用，代码如下。

```
func Run(tty bool, comArray []string, res *subsystems.ResourceConfig,
containerName string) {
```

```
    parent, writePipe := container.NewParentProcess(tty)
    if parent == nil {
        log.Errorf("New parent process error")
        return
    }
    if err := parent.Start(); err != nil {
        log.Error(err)
    }

    //记录容器信息
    containerName, err := recordContainerInfo(parent.Process.Pid, comArray,
containerName)
    if err != nil {
        log.Errorf("Record container info error %v", err)
        return
    }

    cgroupManager := cgroups.NewCgroupManager("mydocker-cgroup")
    defer cgroupManager.Destroy()
    cgroupManager.Set(res)
    cgroupManager.Apply(parent.Process.Pid)

    sendInitCommand(comArray, writePipe)
    if tty {
        parent.Wait()
        deleteContainerInfo(containerName)
    }
}
```

可以看到，如果最后是使用需要创建 tty 方式的容器，那么在容器退出后，就会删除容器的相关信息，实现也很简单，把对应目录的信息都删除就好了。

```
func deleteContainerInfo(containerId string) {
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, containerId)
    if err := os.RemoveAll(dirURL); err != nil {
        log.Errorf("Remove dir %s error %v", dirURL, err)
    }
}
```

到此为止，就完成了信息的收集。容器创建后，所有需要的信息都被存储到“/var/run/mydocker/ 容器名”下，下面就可以通过读取遍历这个目录下的容器去实现 mydocker ps 命令了。

流程图如图 5.2 所示。

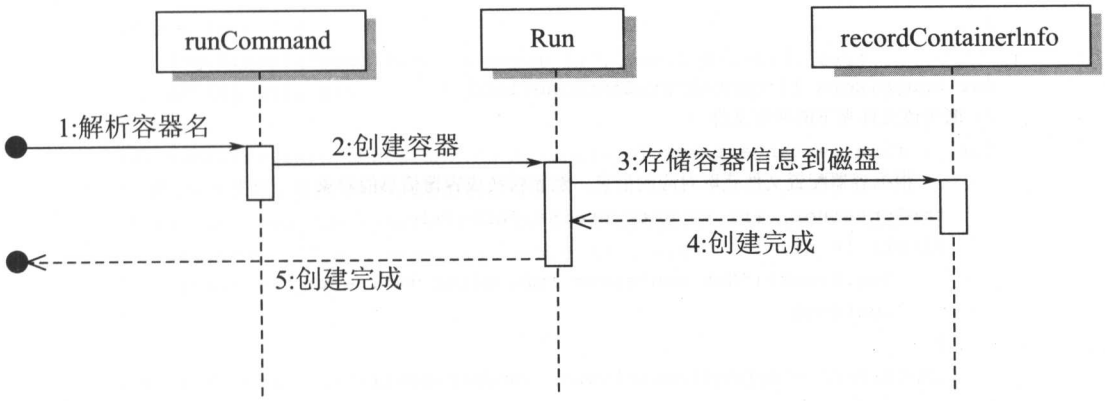


图 5.2

5.2.2　实现 mydocker ps

首先，在许久没有改动过的 main.go 函数里面加一个 listCommand。

```
app.Commands = []cli.Command{
        initCommand,
        runCommand,
        listCommand,
    }

var listCommand = cli.Command{
    Name:  "ps",
    Usage: "list all the containers",
    Action: func(context *cli.Context) error {
        ListContainers()
        return nil
    },
}
```

具体实现也很简单，如下。

```
func ListContainers() {
    // 找到存储容器信息的路径 /var/run/mydocker
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, "")
    dirURL = dirURL[:len(dirURL)-1]
    // 读取该文件夹下的所有文件
    files, err := ioutil.ReadDir(dirURL)
    if err != nil {
        log.Errorf("Read dir %s error %v", dirURL, err)
        return
```

```
    }
    var containers []*container.ContainerInfo
    //遍历该文件夹下的所有文件
    for _, file := range files {
        //根据容器配置文件获取对应的信息，然后转换成容器信息的对象
        tmpContainer, err := getContainerInfo(file)
        if err != nil {
            log.Errorf("Get container info error %v", err)
            continue
        }
        containers = append(containers, tmpContainer)
    }

    //使用 tabwriter.NewWriter 在控制台打印出容器信息
    // tabwriter 是引用的 text/tabwriter 类库，用于在控制台打印对齐的表格
    w := tabwriter.NewWriter(os.Stdout, 12, 1, 3, ' ', 0)
    //控制台输出的信息列
    fmt.Fprint(w, "ID\tNAME\tPID\tSTATUS\tCOMMAND\tCREATED\n")
    for _, item := range containers {
        fmt.Fprintf(w, "%s\t%s\t%s\t%s\t%s\t%s\n",
            item.Id,
            item.Name,
            item.Pid,
            item.Status,
            item.Command,
            item.CreatedTime)
    }
    //刷新标准输出流缓存区，将容器列表打印出来
    if err := w.Flush(); err != nil {
        log.Errorf("Flush error %v", err)
        return
    }
}


func getContainerInfo(file os.FileInfo) (*container.ContainerInfo, error) {
    //获取文件名
    containerName := file.Name()
    //根据文件名生成文件绝对路径
    configFileDir := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    configFileDir = configFileDir + container.ConfigName
    //读取 config.json 文件内的容器信息
    content, err := ioutil.ReadFile(configFileDir)
```

```
    if err != nil {
        log.Errorf("Read file %s error %v", configFileDir, err)
        return nil, err
    }
    var containerInfo container.ContainerInfo
    // 将 json 文件信息反序列化成容器信息对象
    if err := json.Unmarshal(content, &containerInfo); err != nil {
        log.Errorf("Json unmarshal error %v", err)
        return nil, err
    }

    return &containerInfo, nil
}
```

流程图如图 5.3 所示。

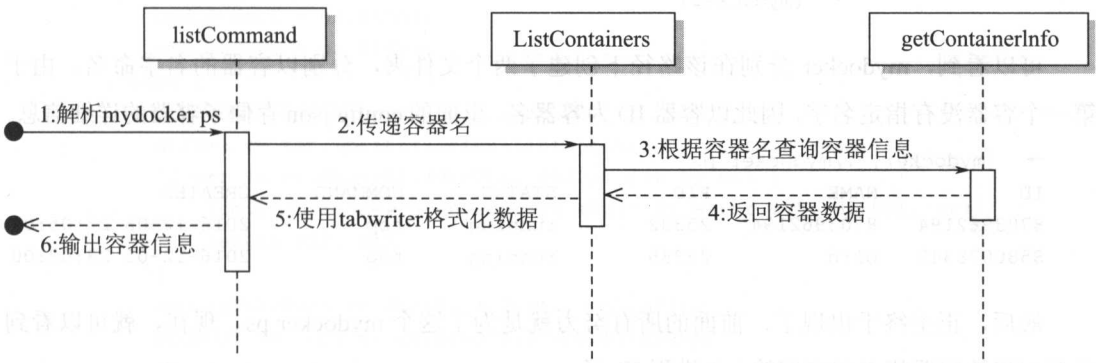


图 5.3

到此为止，我们就实现了所有 mydocker ps 需要的步骤。下面编译运行一下试试看。

首先，创建一个不指定容器名的 detach 容器；然后，再创建一个指定容器名的 detach 容器。

```
→  [mydocker] ./mydocker run -d top
{"level":"info","msg":"createTty false","time":"2016-12-05T23:05:02+08:00"}
{"level":"info","msg":"command all is top","time":"2016-12-05T23:05:02+08:00"}
→  [mydocker] ./mydocker run -d --name bird top
{"level":"info","msg":"createTty false","time":"2016-12-05T23:08:00+08:00"}
{"level":"info","msg":"command all is top","time":"2016-12-05T23:08:00+08:00"}
→  [mydocker]
```

接下来，查看一下存储容器信息的文件结构。

```
→  [mydocker] pwd
/var/run/mydocker
```

```
→  [mydocker] ll
total 0
drw------- 2 root root 60 Dec  5 23:05 8703962194
drw------- 2 root root 60 Dec  5 23:08 bird
→  [mydocker] tree
.
├── 8703962194
│   └── config.json
└── bird
    └── config.json

2 directories, 2 files
→  [mydocker] cat bird/config.json
{"pid":"25385","id":"8580078345","name":"bird","command":"top","createTi
me":"2016-12-05 23:08:00","status":"running"}
               →  [mydocker]
```

可以看到，mydocker 分别在该路径下创建了两个文件夹，分别以容器的名字命名。由于第一个容器没有指定名字，因此以容器 ID 为容器名。里面的 config.json 存储了容器的详细信息。

```
→  [mydocker] ./mydocker ps
ID          NAME        PID         STATUS      COMMAND     CREATED
8703962194  8703962194  25332       running     top         2016-12-05 23:05:02
8580078345  bird        25385       running     top         2016-12-05 23:08:00
```

最后，正主终于出现了，前面的所有努力就是为了这个 mydocker ps。现在，就可以看到所有创建的容器状态和它们的 init 进程 ID 了。

5.3 实现查看容器日志

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.3
```

5.2 节实现了 mydocker ps 命令，可以查看处于后台运行中的容器。那么，还有一个问题就是，如何查看处于后台运行中的容器日志。一般来说，对于容器中运行的进程，使日志达到标准输出是一个非常好的实现方案，因此需要将容器中的标准输出保存下来，以便需要的时候访问。我们就以此作为思路来实现 mydocker logs 命令。

我们会将容器进程的标准输出挂载到“/var/run/mydocker/ 容器名 /container.log”文件中，这样就可以在调用 mydocker logs 的时候去读取这个文件，并将进程内的标准输出打印出来。

首先，需要修改一下原来的实现，在创建容器的时候，把进程的标准输出重新定向一下。

```
func NewParentProcess(tty bool, containerName string) (*exec.Cmd, *os.File) {
    readPipe, writePipe, err := NewPipe()
    if err != nil {
        log.Errorf("New pipe error %v", err)
        return nil, nil
    }
    cmd := exec.Command("/proc/self/exe", "init")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_
            NEWNS | syscall.CLONE_NEWNET | syscall.CLONE_NEWIPC,
    }

    if tty {
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
    } else {
       // 生成容器对应目录的 container.log 文件
        dirURL := fmt.Sprintf(DefaultInfoLocation, containerName)
        if err := os.MkdirAll(dirURL, 0622); err != nil {
            log.Errorf("NewParentProcess mkdir %s error %v", dirURL, err)
            return nil, nil
        }
        stdLogFilePath := dirURL + ContainerLogFile
        stdLogFile, err := os.Create(stdLogFilePath)
        if err != nil {
           log.Errorf("NewParentProcess create file %s error %v", stdLogFilePath,
err)
            return nil, nil
        }
        // 把生成好的文件赋值给 stdout，这样就能把容器内的标准输出重定向到这个文件中
        cmd.Stdout = stdLogFile
    }

    cmd.ExtraFiles = []*os.File{readPipe}
    cmd.Dir = "/root/busybox"
    return cmd, writePipe
}
```

下面在 main.go 里面添加一条命令。

```
app.Commands = []cli.Command{
       initCommand,
```

```
        runCommand,
        listCommand,
        logCommand,
    }
```

定义如下。

```
var logCommand = cli.Command{
    Name: "logs",
    Usage: "print logs of a container",
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Please input your container name")
        }
        containerName := context.Args().Get(0)
        logContainer(containerName)
        return nil
    },
}
```

此处会判断一下输出参数，因为要根据这个输出参数到对应的文件夹中寻找日志文件，所以如果不指定或者指定错误都是会报错的。具体的日志查看方法还是比较简单的，如下。

```
func logContainer(containerName string) {
    // 找到对应文件夹的位置
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    logFileLocation := dirURL + container.ContainerLogFile
    // 打开日志文件
    file, err := os.Open(logFileLocation)
    defer file.Close()
    if err != nil {
        log.Errorf("Log container open file %s error %v", logFileLocation, err)
        return
    }
    // 将文件内的内容都读取出来
    content, err := ioutil.ReadAll(file)
    if err != nil {
        log.Errorf("Log container read file %s error %v", logFileLocation, err)
        return
    }
    // 使用 fmt.Fprint 函数将读出来的内容输入到标准输出，也就是控制台上
    fmt.Fprint(os.Stdout, string(content))
}
```

下面来实验一下，首先以 detach 的方式创建一个容器。

```
→  [mydocker] ./mydocker run -d --name bird top
{"level":"info","msg":"createTty false","time":"2016-12-08T20:41:47+08:00"}
{"level":"info","msg":"command all is top","time":"2016-12-08T20:41:47+08:00"}
```

可以在对应的文件夹下看到，container.log 文件已经生成了。

```
→  [/var/run/mydocker] tree
.
└── bird
    ├── config.json
    └── container.log

1 directory, 2 files
```

来看一下容器的 ps 输出，如下。

```
→  [mydocker] ./mydocker ps
ID           NAME        PID         STATUS      COMMAND     CREATED
1084902341   bird        2000        running     top         2016-12-08 20:41:47
```

运行一下 logs 命令。

```
→  [mydocker] ./mydocker logs bird
Mem: 264484K used, 237248K free, 1256K shrd, 19616K buff, 115284K cached
CPU:   0% usr   0% sys   0% nic 100% idle   0% io   0% irq   0% sirq
Load average: 0.06 0.04 0.05 2/147 3
PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
```

可以看到，mydocker logs 命令已经成功运行并输出容器的日志。

5.4　实现进入容器 Namespace

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.4
```

通过 5.3 节的学习，我们已经可以查看后台运行的容器日志了。但是容器一旦创建后，就无法再次进入容器，因此我们需要能够再次进入到容器内部的功能，这就是这一节要实现的 mydocker exec 的功能。

5.4.1 setns

setns 是一个系统调用，可以根据提供的 PID 再次进入到指定的 Namespace 中。它需要先打开 /proc/[pid]/ns/ 文件夹下对应的文件，然后使当前进程进入到指定的 Namespace 中。系统调用描述非常简单，但是有一点对于 Go 来说很麻烦。对于 Mount Namespace 来说，一个具有多线程的进程是无法使用 setns 调用进入到对应的命名空间的。但是，Go 每启动一个程序就会进入多线程状态，因此无法简简单单地在 Go 里面直接调用系统调用，使当前的进程进入对应的 Mount Namespace。这里需要借助 C 来实现这个功能。

5.4.2 Cgo

Cgo 是一个很炫酷的功能，允许 Go 程序去调用 C 的函数与标准库。你只需要以一种特殊的方式在 Go 的源代码里写出需要调用的 C 的代码，Cgo 就可以把你的 C 源码文件和 Go 文件整合成一个包。下面举一个最简单的例子，在这个例子中有两个函数——Random 和 Seed，在它们里面调用了 C 的 random 和 srandom 函数。

```
package rand

/*
#include <stdlib.h>
*/
import "C"

func Random() int {
    return int(C.random())
}

func Seed(i int) {
    C.srandom(C.uint(i))
}
```

这段代码导入了 C，但是你会发现在 Go 标准库里面并没有这个包，那是因为这根本就不是一个真正的包，而只是 Cgo 创建的一个特殊命名空间，用来与 C 的命名空间交流。这两个函数都分别调用了 C 里面的 random 和 uint 函数，然后对它们进行了类型转换。这就实现了 Go 代码里面调用 C 的功能。

5.4.3 实现命令

前面说了那么多，下面开始来真正实现这个功能。首先，需要使用 C 根据指定的 PID 进

入对应的命名空间。

```
package nsenter

/*
#include <errno.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

// 这里的 __attribute__((constructor)) 指的是，一旦这个包被引用，那么这个函数就会被自动执行
// 类似于构造函数，会在程序一启动的时候运行
__attribute__((constructor)) void enter_namespace(void) {
    char *mydocker_pid;
    // 从环境变量中获取需要进入的 PID
    mydocker_pid = getenv("mydocker_pid");
    if (mydocker_pid) {
        //fprintf(stdout, "got mydocker_pid=%s\n", mydocker_pid);
    } else {
        //fprintf(stdout, "missing mydocker_pid env skip nsenter");
        // 这里，如果没有指定 PID，就不需要向下执行，直接退出
        return;
    }
    char *mydocker_cmd;
    // 从环境变量里面获取需要执行的命令
    mydocker_cmd = getenv("mydocker_cmd");
    if (mydocker_cmd) {
        //fprintf(stdout, "got mydocker_cmd=%s\n", mydocker_cmd);
    } else {
        //fprintf(stdout, "missing mydocker_cmd env skip nsenter");
        // 如果没有指定命令，则直接退出
        return;
    }
    int i;
    char nspath[1024];
    // 需要进入的五种 Namespace
    char *namespaces[] = { "ipc", "uts", "net", "pid", "mnt" };

    for (i=0; i<5; i++) {
        // 拼接对应的路径 /proc/pid/ns/ipc，类似这样
        sprintf(nspath, "/proc/%s/ns/%s", mydocker_pid, namespaces[i]);
        int fd = open(nspath, O_RDONLY);
        // 这里才真正调用 setns 系统调用进入对应的 Namespace
```

```
        if (setns(fd, 0) == -1) {
            //fprintf(stderr, "setns on %s namespace failed: %s\n", namespaces[i],
strerror(errno));
        } else {
            //fprintf(stdout, "setns on %s namespace succeeded\n", namespaces[i]);
        }
        close(fd);
    }
    // 在进入的 Namespace 中执行指定的命令
    int res = system(mydocker_cmd);
    //退出
    exit(0);
    return;
}
*/
import "C"
```

可以看到，这段程序还是很怪异的，和普通的 Go 代码是不一样的。这里主要使用了构造函数，然后导入了 C 模块，一旦这个包被引用，它就会在所有 Go 运行的环境启动之前执行，这样就避免了 Go 多线程导致的无法进入 mnt Namespace 的问题。这段程序执行完毕后，Go 程序才会执行。

但是这会带来一个问题，就是只要这个包被导入，它就会在所有 Go 代码前执行，那么即使那些不需要使用 exec 这段代码的地方也会运行这段程序。举例来说，使用 mydocker run 来创建容器，但是这段 C 代码依然会执行，这就会影响前面已经完成的功能。因此，需要在这段 C 代码前面一开始的位置就指定环境变量，对于不使用 exec 功能的 Go 代码，只要不设置对应的环境变量，那么当 C 程序检测到没有这个环境变量时，就会直接退出，继续执行原来的代码，并不会影响原来的逻辑。

下面就需要增加代码来实现 exec 的功能。首先看一下 execCommand 的定义，如下。

```
var execCommand = cli.Command{
    Name: "exec",
    Usage: "exec a command into container",
    Action: func(context *cli.Context) error {
        // 这句代码下面会提及
        //This is for callback
        if os.Getenv(ENV_EXEC_PID) != "" {
            log.Infof("pid callback pid %s", os.Getgid())
            return nil
        }
```

```
        // 我们希望命令格式是 mydocker exec 容器名 命令
        if len(context.Args()) < 2 {
            return fmt.Errorf("Missing container name or command")
        }
        containerName := context.Args().Get(0)
        var commandArray []string
        // 将除了容器名之外的参数当作需要执行的命令处理
        for _, arg := range context.Args().Tail() {
            commandArray = append(commandArray, arg)
        }
        // 执行命令
        ExecContainer(containerName, commandArray)
        return nil
    },
}
```

这里主要是将获取到的容器名和需要的命令处理完成后，交给下面的函数，下面看一下 ExecContainer 的实现。

```
// 这里是根据提供的容器名获取对应容器的 PID
func getContainerPidByName(containerName string) (string, error) {
    // 先拼接出存储容器信息的路径
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    configFilePath := dirURL + container.ConfigName
    // 读取该对应路径下的文件内容
    contentBytes, err := ioutil.ReadFile(configFilePath)
    if err != nil {
        return "", err
    }
    var containerInfo container.ContainerInfo
    // 将文件内容反序列化成容器信息对象，然后返回对应的 PID
    if err := json.Unmarshal(contentBytes, &containerInfo); err != nil {
        return "", err
    }
    return containerInfo.Pid, nil
}
```

下面是主要实现。

```
/*
前面的 C 代码里已经出现 mydocker_pid 和 mydocker_cmd 这两个 Key，主要是为了控制是否执行 C 代
码里面的 setns。
*/
const ENV_EXEC_PID = "mydocker_pid"
const ENV_EXEC_CMD = "mydocker_cmd"
```

```
func ExecContainer(containerName string, comArray []string) {
    //根据传递过来的容器名获取宿主机对应的 PID
    pid, err := getContainerPidByName(containerName)
    if err != nil {
            log.Errorf("Exec container getContainerPidByName %s error %v",
containerName, err)
        return
    }
    //把命令以空格为分隔符拼接成一个字符串，便于传递
    cmdStr := strings.Join(comArray, " ")
    log.Infof("container pid %s", pid)
    log.Infof("command %s", cmdStr)

    //这里是重点，下面会讲解
    cmd := exec.Command("/proc/self/exe", "exec")
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    os.Setenv(ENV_EXEC_PID, pid)
    os.Setenv(ENV_EXEC_CMD, cmdStr)

    if err := cmd.Run(); err != nil {
        log.Errorf("Exec container %s error %v", containerName, err)
    }
}
```

这里又遇到熟悉的 /proc/self/exe，只不过是换了后面的参数，由原来的 init 变成了现在的 exec。这么做的目的就是为了那段 C 代码的执行。因为一旦程序启动，那段 C 代码就会运行，那么对于我们使用 exec 来说，当容器名和对应的命令传递进来以后，程序已经执行了，而且那段 C 代码也应该运行完毕。那么，怎么指定环境变量让它再执行一遍呢？这里就用到了这个 /proc/self/exe。这里又创建了一个 command，只不过这次只是简单地 fork 出来一个进程，不需要这个进程拥有什么命名空间的隔离，然后把这个进程的标准输入输出都绑定到宿主机上。这样去 run 这里的进程时，实际上就是又运行了一遍自己的程序，但是这时有一点不同的就是，再一次运行的时候已经指定了环境变量，所以 C 代码执行的时候就能拿到对应的环境变量，便可以进入到指定的 Namespace 中进行操作了。这时应该就可以明白前面一段代码的意义了。

```
//This is for callback
if os.Getenv(ENV_EXEC_PID) != "" {
    log.Infof("pid callback pid %s", os.Getgid())
```

```
    return nil
}
```

这里就是第二次进入本程序执行 exec 的时候，如果已经指定了环境变量，说明 C 代码已经运行，直接返回就可以了，以免重复调用。具体的流程图如图 5.4 所示。

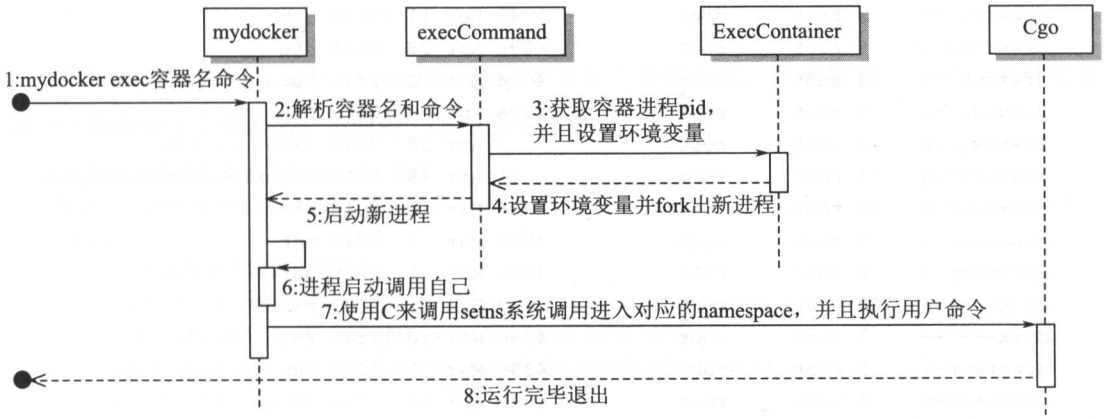


图 5.4

下面来运行一下试试。

```
// 先编译一下当前代码
➜  [mydocker] go build
// 创建一个名字为 bird 的 detach 容器
➜  [mydocker] ./mydocker run --name bird -d top
{"level":"info","msg":"createTty false","time":"2016-12-11T13:33:31+08:00"}
{"level":"info","msg":"command all is top","time":"2016-12-11T13:33:31+08:00"}
// 查看一下当前运行中的容器
➜  [mydocker] ./mydocker ps
ID           NAME        PID         STATUS      COMMAND     CREATED
7034947278   bird        6357        running     top         2016-12-11 13:33:31
//exec 进入容器内部
➜  [mydocker] ./mydocker exec bird sh
{"level":"info","msg":"container pid 6357","time":"2016-12-11T13:34:42+08:00"}
{"level":"info","msg":"command sh","time":"2016-12-11T13:34:42+08:00"}
// 在容器内部执行 ps -ef 可以发现 PID 为 1 的进程为 top，也就意味着已经成功进入到了容器内部
/ # ps -ef
PID   USER     COMMAND
    1 root     top
    4 root     sh -c sh
    5 root     ps -ef
/ #exit
```

```
// 再执行一个会立刻退出的 ls -l 命令，可以看到当前程序工作目录中的内容都被列出来了
→  [mydocker] ./mydocker exec bird "ls -l"
{"level":"info","msg":"container pid 6357","time":"2016-12-11T13:37:04+08:00"}
{"level":"info","msg":"command ls -l","time":"2016-12-11T13:37:04+08:00"}
total 56
drwxrwxr-x    2 root     root          4096 Apr 28  2015 bin
drwxr-xr-x    2 root     root            40 Dec 11 05:33 dev
drwxr-xr-x    6 root     root          4096 Apr 28  2015 etc
drwxrwxr-x    3 root     root          4096 Mar  1  2015 home
drwxrwxr-x    2 root     root          4096 Apr 28  2015 lib
lrwxrwxrwx    1 root     root             3 Apr 28  2015 lib64 -> lib
lrwxrwxrwx    1 root     root            11 Apr 28  2015 linuxrc -> bin/busybox
drwxrwxr-x    2 root     root          4096 Mar  1  2015 media
drwxrwxr-x    2 root     root          4096 Mar  1  2015 mnt
drwxrwxr-x    2 root     root          4096 Mar  1  2015 opt
dr-xr-xr-x  100 root     root             0 Dec 11 05:33 proc
drwx------    2 root     root          4096 Nov 30 11:41 root
drwxrwxr-x    2 root     root          4096 Mar  1  2015 run
drwxr-xr-x    2 root     root          4096 Apr 28  2015 sbin
drwxrwxr-x    2 root     root          4096 Mar  1  2015 sys
drwxrwxrwt    3 root     root          4096 Apr 28  2015 tmp
drwxrwxr-x    6 root     root          4096 Apr 28  2015 usr
drwxrwxr-x    4 root     root          4096 Apr 28  2015 var
```

5.5 实现停止容器

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.5
```

一个容器的完整生命周期除了需要 run 创建运行还需要 stop。5.4 节完成了对容器的创建、查看、exec 等操作，下面需要完成容器的停止和清理的一些工作。其实，stop 容器的原理很简单，主要就是查找到它的主进程 PID，然后发送 SIGTERM 信号，等待进程结束就好。下面就来实现它。

首先，来看一下对于 stopCommand 的定义，如下。

```
var stopCommand = cli.Command{
    Name: "stop",
    Usage: "stop a container",
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
```

```
            return fmt.Errorf("Missing container name")
        }
        containerName := context.Args().Get(0)
        stopContainer(containerName)
        return nil
    },
}
```

我们期望的调用方式是“mydocker stop 容器名”这种方式。因此需要在一开始的时候检测一下是否输入了容器名。下面看一下后面的函数。

```
// 根据容器名获取对应的 struct 结构
func getContainerInfoByName(containerName string) (*container.ContainerInfo,
error) {
    // 构造存放容器信息的路径
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    configFilePath := dirURL + container.ConfigName
    contentBytes, err := ioutil.ReadFile(configFilePath)
    if err != nil {
        log.Errorf("Read file %s error %v", configFilePath, err)
        return nil, err
    }
    var containerInfo container.ContainerInfo
    // 将容器信息字符串反序列化成对应的对象
    if err := json.Unmarshal(contentBytes, &containerInfo); err != nil {
        log.Errorf("GetContainerInfoByName unmarshal error %v", err)
        return nil, err
    }
    return &containerInfo, nil
}
```

下面看一下真正的 stopContainer 实现。

```
func stopContainer(containerName string) {
    // 根据容器名获取对应的主进程 PID
    pid, err := GetContainerPidByName(containerName)
    if err != nil {
        log.Errorf("Get contaienr pid by name %s error %v", containerName, err)
        return
    }
    // 将 string 类型的 PID 转换为 int 类型
    pidInt, err := strconv.Atoi(pid)
    if err != nil {
        log.Errorf("Conver pid from string to int error %v", err)
        return
```

```
    }
    // 系统调用 kill 可以发送信号给进程，通过传递 syscall.SIGTERM 信号，去杀掉容器主进程
    if err := syscall.Kill(pidInt, syscall.SIGTERM); err != nil {
        log.Errorf("Stop container %s error %v", containerName, err)
        return
    }
    // 根据容器名获取对应的信息对象
    containerInfo, err := getContainerInfoByName(containerName)
    if err != nil {
        log.Errorf("Get container %s info error %v", containerName, err)
        return
    }
    // 至此，容器进程已经被 kill，所以下面需要修改容器状态，PID 可以置为空
    containerInfo.Status = container.STOP
    containerInfo.Pid = " "
    // 将修改后的信息序列化成 json 的字符串
    newContentBytes, err := json.Marshal(containerInfo)
    if err != nil {
        log.Errorf("Json marshal %s error %v", containerName, err)
        return
    }
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    configFilePath := dirURL + container.ConfigName
    // 重新写入新的数据覆盖原来的信息
    if err := ioutil.WriteFile(configFilePath, newContentBytes, 0622); err != nil {
        log.Errorf("Write file %s error", configFilePath, err)
    }
}
```

stopContainer 的主要步骤如下。

1. 获取容器 PID。
2. 对该 PID 发送 kill 信号。
3. 修改容器信息。
4. 重新写入存储容器信息的文件。

流程图如图 5.5 所示。

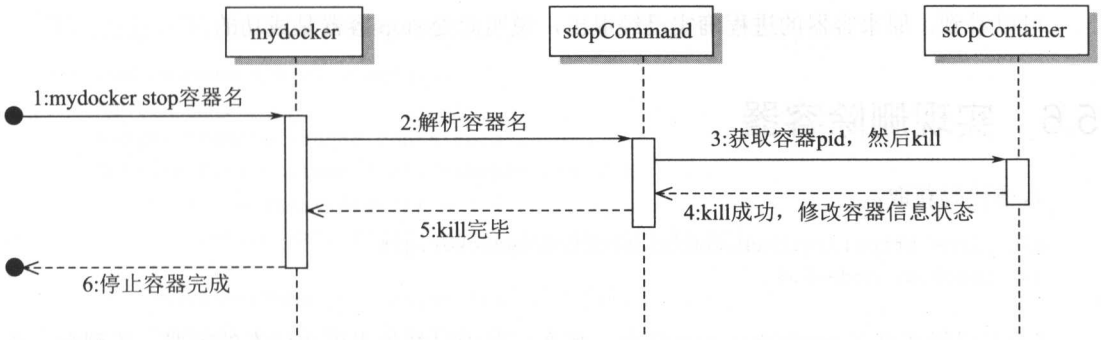


图 5.5

下面进行编译看一下效果。

```
→  [mydocker] go build
// 创建一个名为 bird 的 detach 容器
→  [mydocker] ./mydocker run --name bird -d top
{"level":"info","msg":"createTty false","time":"2016-12-11T18:00:08+08:00"}
{"level":"info","msg":"command all is top","time":"2016-12-11T18:00:08+08:00"}
// 这时 mydocker ps 可以看到该容器信息，状态为 running
→  [mydocker] ./mydocker ps
ID           NAME         PID         STATUS      COMMAND     CREATED
2230122118   bird         7887        running     top         2016-12-11 18:00:08
```

此时，在宿主机上执行 ps -ef，看一下这个容器的进程。

```
→  [mydocker] ps -ef | grep top
root      7887     1   0 18:00 pts/0    00:00:00 top
root      7902  1948   0 18:00 pts/1    00:00:00 grep --color=auto --exclude-dir=.bzr
--exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn top
```

可以看到，我们能够发现这个容器 PID 为 7887 的主进程，下面尝试去 stop 它。

```
→  [mydocker] ./mydocker stop bird
// 可以看到，stop 后，容器的 PID 已经消失，而且状态变为 stopped
→  [mydocker] ./mydocker ps
ID           NAME         PID         STATUS      COMMAND     CREATED
2230122118   bird                     stopped     top         2016-12-11 18:00:08
```

此时，在宿主机上执行 ps -ef，看一下效果。

```
→  [mydocker] ps -ef | grep top
root      7925 1948   0 18:03 pts/1    00:00:00 grep --color=auto --exclude-dir=.bzr
--exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn top
```

可以看到，原来容器的进程确实已经退出，说明此处 stop 容器是成功的。

5.6 实现删除容器

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.6
```

5.5 节已经实现了 mydocker stop 命令，那么，对于已经处于停止状态的容器，还剩余一个操作来完成容器的生命周期。进行 remove 操作之后，就可以把整个容器的生命周期完结了。本节就会完成这最后一步的清理工作。

mydocker rm 实现起来很简单，主要是文件操作，因为容器对应的进程已经被停止，所以只需要将对应记录文件信息的地址删除就可以了，下面来看一下实现。

```
func removeContainer(containerName string) {
    //根据容器名获取容器对应的信息
    containerInfo, err := getContainerInfoByName(containerName)
    if err != nil {
        log.Errorf("Get container %s info error %v", containerName, err)
        return
    }
    //只删除处于停止状态的容器
    if containerInfo.Status != container.STOP {
        log.Errorf("Couldn't remove running container")
        return
    }
    //找到对应存储容器信息的文件路径
    dirURL := fmt.Sprintf(container.DefaultInfoLocation, containerName)
    //将所有信息包括子目录都移除
    if err := os.RemoveAll(dirURL); err != nil {
        log.Errorf("Remove file %s error %v", dirURL, err)
        return
    }
}
```

可以看到，删除容器的动作很简单，主要分为以下 4 个步骤。

1. 根据容器名查找容器信息。
2. 判断容器是否处于停止状态。
3. 查找容器存储信息的地址。
4. 移除记录容器信息的文件。

下面再看一下 removeCommand 的定义。

```
var removeCommand = cli.Command{
    Name: "rm",
    Usage: "remove unused containers",
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 1 {
            return fmt.Errorf("Missing container name")
        }
        containerName := context.Args().Get(0)
        removeContainer(containerName)
        return nil
    },
}
```

希望的调用方式为“mydocker rm 容器名”。下面就来实验一下效果。

```
// 编译代码
→  [mydocker] go build
// 创建 detach 的容器
→  [mydocker] ./mydocker run --name bird -d top
{"level":"info","msg":"createTty false","time":"2016-12-11T19:42:14+08:00"}
{"level":"info","msg":"command all is top","time":"2016-12-11T19:42:14+08:00"}
// 查询发现容器处于运行状态
→  [mydocker] ./mydocker ps
ID          NAME        PID         STATUS      COMMAND     CREATED
1675819994  bird        8520        running     top         2016-12-11 19:42:14
// 删除处于运行状态的容器失败，说明我们设置为不能删除运行态容器是正确的
→  [mydocker] ./mydocker rm bird
{"level":"error","msg":"Couldn't remove running
container","time":"2016-12-11T19:42:25+08:00"}

// 停止容器
→  [mydocker] ./mydocker stop bird
// 移除停止的容器
→  [mydocker] ./mydocker rm bird
// 再次查询，发现容器的信息已经不存在了
→  [mydocker] ./mydocker ps
ID          NAME        PID         STATUS      COMMAND     CREATED
```

5.7　实现通过容器制作镜像

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.7
```

在本章中，通过使用 mydocker run -d 命令实现了同时运行多个容器的功能。这样一来，存在的问题就是使用一个镜像启动的多个容器会使用同一个 AUFS 文件系统，容器的可写层会互相影响。

所以，本节要实现如下两个目的。

- 为每个容器分配单独的隔离文件系统。
- 修改 mydocker commit 命令，实现对不同容器进行打包镜像的功能。

首先，为每个容器分配单独的隔离文件系统。

修改 main_command.go 文件，如下。

1. 在 runCommand 命令中添加 imageName 作为第一个参数输入。
2. 调用 Run 函数时，传参列表添加 imageName。

```
//get image name
imageName := cmdArray[0]
cmdArray = cmdArray[1:]

//add imageName
Run(createTty, cmdArray, resConf, containerName, volume, imageName)
```

修改 run.go 文件，如下。

1. 修改 Run 方法：
   - 在 Run 函数的参数列表中添加 imageName。
   - 调用 container.NewParentProcess 函数的传参列表，其中有 tty、containerName、volume 和 imageName。
   - 调用 container.DeleteWorkSpace 函数的传参列表，其中有 volume、imageName 和 containerName。

```
func Run(tty bool, comArray []string, res *subsystems.ResourceConfig,
containerName, volume, imageName string) {
...
    parent, writePipe := container.NewParentProcess(tty, containerName, volume,
    imageName)
...
     container.DeleteWorkSpace(volume, imageName, containerName)
```

2. 在 recordContainerInfo 函数的参数列表中增加 volume。在 containerInfo 结构体中添加存

储配置信息 ContainerInfo.Volume。

```
func recordContainerInfo(containerPID int, commandArray []string, containerName,
id, volume string) (string, error) {
    ...
    containerInfo := &container.ContainerInfo{
        Id:          id,
        Pid:         strconv.Itoa(containerPID),
        Command:     command,
        CreatedTime: createTime,
        Status:      container.RUNNING,
        Name:        containerName,
        Volume:      volume,
    }
    ...
}
```

修改 container_process.go 文件，如下。

1. 将 RootUrl、MntUrl 和 WriteLayerUrl 添加为全局变量。
2. 在 NewParentProcess 函数中添加 imageName 参数。
3. 修改 NewWorkSpace 函数的传参列表为 volume、imageName 和 containerName。
4. 在 ContainerInfo 中添加 volume 成员。

```
// 将 RootUrl、MntUrl 和 WriteLayerUrl 添加为全局变量
var (
    ...
    RootUrl             string = "/root"
    MntUrl              string = "/root/mnt/%s"
    WriteLayerUrl       string = "/root/writeLayer/%s"
)

func NewParentProcess(tty bool, containerName, volume, imageName string) (*exec.
Cmd, *os.File) {
...
    NewWorkSpace(volume, imageName, containerName)
    cmd.Dir = fmt.Sprintf(MntUrl, containerName)
...
}

// 存储 volume 信息
type ContainerInfo struct {
    ...
    Volume      string `json:"volume"`
}
```

添加 volume.go 文件，把对 Volume 的操作从 container_process.go 文件中抽出来写到 volume.go 文件中，然后进行如下修改。

- NewWorkSpace 函数的作用是为每个容器创建文件系统。将 NewWorkSpace 函数的参数列表修改为 volume、imageName 和 containerName。

```
func NewWorkSpace(volume, imageName, containerName string) {
    ...
    CreateReadOnlyLayer(imageName)
    CreateWriteLayer(containerName)
    CreateMountPoint(containerName, imageName)
    ...
    MountVolume(volumeURLs, containerName)
}
```

- CreateReadOnlyLayer 函数的作用是根据用户输入的镜像为每个容器创建只读层。将 CreateReadOnlyLayer 函数的参数修改为 imageName，镜像解压出来的只读层以 RootUrl + imageName 命名。

```
// 解压 tar 格式的镜像文件作为只读层
func CreateReadOnlyLayer(imageName string) error {
    unTarFolderUrl := RootUrl + "/" + imageName + "/"
    imageUrl := RootUrl + "/" + imageName + ".tar"
    exist, err := PathExists(unTarFolderUrl)
    if err != nil {
        log.Infof("Fail to judge whether dir %s exists. %v", unTarFolderUrl, err)
        return err
    }

    if !exist {
        if err := os.MkdirAll(unTarFolderUrl, 0622); err != nil {
            log.Errorf("Mkdir %s error %v", unTarFolderUrl, err)
            return err
        }

        if _, err := exec.Command("tar", "-xvf", imageUrl, "-C",
unTarFolderUrl).CombinedOutput(); err != nil {
            log.Errorf("Untar dir %s error %v", unTarFolderUrl, err)
            return err
        }
    }
    return nil
}
```

- CreateWriteLayer 函数的作用是为每个容器创建一个读写层。将 CreateWriteLayer 函数的参数修改成 containerName，容器的读写层修改成以 WriteLayerUrl + containerName 命名。

```
//create writeLayer for each container
func CreateWriteLayer(containerName string) {
    writeURL := fmt.Sprintf(WriteLayerUrl, containerName)
    if err := os.MkdirAll(writeURL, 0777); err != nil {
        log.Infof("Mkdir write layer dir %s error. %v", writeURL, err)
    }
}
```

- MountVolume 函数的作用是根据用户输入的 volume 参数获取相应要挂载的宿主机数据卷 URL 和容器中的挂载点 URL，并挂载数据卷。MountVolume 函数的参数列表改为 volumeURLs 和 containerName，容器的挂载点改为以 MntUrl + containerName + containerUrl 命名。

```
//mount volume
func MountVolume(volumeURLs []string, containerName string) error {
    parentUrl := volumeURLs[0]
    if err := os.Mkdir(parentUrl, 0777); err != nil {
        log.Infof("Mkdir parent dir %s error. %v", parentUrl, err)
    }
    containerUrl := volumeURLs[1]
    mntURL := fmt.Sprintf(MntUrl, containerName)
    containerVolumeURL := mntURL + "/" +  containerUrl
    if err := os.Mkdir(containerVolumeURL, 0777); err != nil {
        log.Infof("Mkdir container dir %s error. %v", containerVolumeURL, err)
    }
    dirs := "dirs=" + parentUrl
    _, err := exec.Command("mount", "-t", "aufs", "-o", dirs, "none",
containerVolumeURL).CombinedOutput()
    if err != nil {
        log.Errorf("Mount volume failed. %v", err)
        return err
    }
    return nil
}
```

- CreateMountPoint 函数的作用是创建容器的根目录，然后把镜像只读层和容器读写层挂载到容器根目录，成为容器的文件系统。CreateMountPoint 函数的参数列表改为 containerName 和 imageName。把通过镜像解压出来的只读层和容器的可读写层用

AUFS 联合挂载成为容器的文件系统。

```
func CreateMountPoint(containerName , imageName string) error {
    mntUrl := fmt.Sprintf(MntUrl, containerName)
    if err := os.MkdirAll(mntUrl, 0777); err != nil {
        log.Errorf("Mkdir mountpoint dir %s error. %v", mntUrl, err)
        return err
    }
    tmpWriteLayer := fmt.Sprintf(WriteLayerUrl, containerName)
    tmpImageLocation := RootUrl + "/" + imageName
    mntURL := fmt.Sprintf(MntUrl, containerName)
    dirs := "dirs=" + tmpWriteLayer + ":" + tmpImageLocation
    _, err := exec.Command("mount", "-t", "aufs", "-o", dirs, "none", mntURL).
CombinedOutput()
    if err != nil {
        log.Errorf("Run command for creating mount point failed %v", err)
        return err
    }
    return nil
}
```

- DeleteWorkSpace 函数的作用是当容器退出时，删除容器的相关文件系统。将 DeleteWorkSpace 函数的参数列表改为 containerName 和 volume。

```
//Delete the AUFS filesystem while container exit
func DeleteWorkSpace(volume, containerName string) {
    if volume != "" {
        volumeURLs := strings.Split(volume, ":")
        length := len(volumeURLs)
        if length == 2 && volumeURLs[0] != "" && volumeURLs[1] != "" {
            DeleteMountPointWithVolume(volumeURLs, containerName)
        } else {
            DeleteMountPoint(containerName)
        }
    } else {
        DeleteMountPoint(containerName)
    }
    DeleteWriteLayer(containerName)
}
```

- DeleteMountPoint 函数的作用是删除未挂载数据卷的容器文件系统。将 DeleteMountPoint 函数的参数修改为 containerName。

```
func DeleteMountPoint(containerName string) error {
    mntURL := fmt.Sprintf(MntUrl, containerName)
```

```
    _, err := exec.Command("umount", mntURL).CombinedOutput()
    if err != nil {
        log.Errorf("Unmount %s error %v", mntURL, err)
        return err
    }
    if err := os.RemoveAll(mntURL); err != nil {
        log.Errorf("Remove mountpoint dir %s error %v", mntURL, err)
        return err
    }
    return nil
}
```

➢ DeleteMountPointWithVolume 函数的作用是删除挂载数据卷容器的文件系统。将 DeleteMountPointWithVolume 的参数列表修改为 volumeURLs 和 containerName。

```
func DeleteMountPointWithVolume(volumeURLs []string, containerName string) error {
    mntURL := fmt.Sprintf(MntUrl, containerName)
    containerUrl := mntURL + "/" +  volumeURLs[1]
    if _, err := exec.Command("umount", containerUrl).CombinedOutput(); err != nil {
        log.Errorf("Umount volume %s failed. %v", containerUrl, err)
        return err
    }

    if _, err := exec.Command("umount", mntURL).CombinedOutput(); err != nil {
        log.Errorf("Umount mountpoint %s failed. %v", mntURL, err)
        return err
    }

    if err := os.RemoveAll(mntURL); err != nil {
        log.Errorf("Remove mountpoint dir %s error %v", mntURL, err)
    }

    return nil
}
```

➢ DeleteWriteLayer 函数的作用是删除容器的读写层。将 DeleteWriteLayer 的参数列表修改为 containerName。

```
func DeleteWriteLayer(containerName string) {
    writeURL := fmt.Sprintf(WriteLayerUrl, containerName)
    if err := os.RemoveAll(writeURL); err != nil {
        log.Infof("Remove writeLayer dir %s error %v", writeURL, err)
    }
}
```

至此，便完成了为每个容器分配单独的隔离文件系统的工作。下面介绍对 mydocker commit 命令的改造，实现对不同容器进行打包镜像的功能。

首先，在 main_command.go 文件中修改 commitCommand。将用户输入参数改为 containerName 和 imageName。调用 commitContainer 方法实现 commit 操作。

```
var commitCommand = cli.Command{
    Name:  "commit",
    Usage: "commit a container into image",
    Action: func(context *cli.Context) error {
        if len(context.Args()) < 2 {
            return fmt.Errorf("Missing container name and image name")
        }
        containerName := context.Args().Get(0)
        imageName := context.Args().Get(1)
        commitContainer(containerName, imageName)
        return nil
    },
}
```

接下来，修改 commit.go 文件中的 commitContainer 函数，实现根据传入的 containerName 制作 imageName.tar 镜像。

```
// 用子目录集合制作 {imageName}.tar 的镜像
func commitContainer(containerName, imageName string) {
    mntURL := fmt.Sprintf(container.MntUrl, containerName)
    mntURL += "/"

    imageTar := container.RootUrl + "/" + imageName + ".tar"

     if _, err := exec.Command("tar", "-czf", imageTar, "-C", mntURL, ".").
CombinedOutput(); err != nil {
        log.Errorf("Tar folder %s error %v", mntURL, err)
    }
}
```

下面来测试一下代码的正确性。首先，用 busybox.tar 镜像启动两个容器 container1 和 container2。container1 容器把宿主机 /root/from1 挂载到容器 /to1 目录下。container2 容器把宿主机 /root/from2 挂载到容器 /to2 目录下。

```
// 启动 container1
./mydocker run -d --name container1 -v /root/from1:/to1 busybox top
{"level":"info","msg":"createTty false","time":"2017-01-07T18:14:01+08:00"}
{"level":"info","msg":"NewWorkSpace volume urls [\"/root/from1\" \"/
```

```
to1\"]","time":"2017-01-07T18:14:01+08:00"}
{"level":"info","msg":"command all is top","time":"2017-01-07T18:14:01+08:00"}
{"level":"warning","msg":"remove cgroup fail remove /sys/fs/cgroup/
memory/mydocker-cgroup/memory.kmem.tcp.max_usage_in_bytes: operation not
permitted","time":"2017-01-07T18:14:01+08:00"}
{"level":"warning","msg":"remove cgroup fail remove /sys/fs/cgroup/cpu/mydocker-
cgroup/cpu.stat: operation not permitted","time":"2017-01-07T18:14:01+08:00"}

// 启动 container2
./mydocker run -d --name container2 -v /root/from2:/to2 busybox top
{"level":"info","msg":"createTty false","time":"2017-01-07T18:16:56+08:00"}
{"level":"info","msg":"NewWorkSpace volume urls [\"/root/from2\" \"/
to2\"]","time":"2017-01-07T18:16:56+08:00"}
{"level":"info","msg":"command all is top","time":"2017-01-07T18:16:56+08:00"}
{"level":"warning","msg":"remove cgroup fail remove /sys/fs/cgroup/
memory/mydocker-cgroup/memory.kmem.tcp.max_usage_in_bytes: operation not
permitted","time":"2017-01-07T18:16:56+08:00"}
{"level":"warning","msg":"remove cgroup fail remove /sys/fs/cgroup/cpu/mydocker-
cgroup/cpu.stat: operation not permitted","time":"2017-01-07T18:16:56+08:00"}

// 查看容器
 ./mydocker ps
ID           NAME         PID         STATUS      COMMAND     CREATED
3526948179   container1   1441        running     top         2017-01-07 18:14:01
1735876290   container2   1480        running     top         2017-01-07 18:16:56
```

另外，打开一个 terminal 创建，查看宿主机 /root 目录下的内容，发现多了 from1 和 from2 两个挂载文件、mnt 这个所有容器的文件系统总入口，以及所有容器读写层的总入口 writeLayer 目录。在 mnt 和 writeLayer 的目录下，都分别创建了 container1 和 container2 两个子目录。mnt/containerName 目录就是整个容器的文件系统。writeLayer/{containerName} 是容器的可读写层，可以看到，里面还有挂载数据卷到容器的挂载点目录。

```
ls /root
busybox  busybox.tar  from1  from2  mnt  writeLayer
ls /root/mnt
container1  container2

ls /root/mnt/container1
bin  dev  etc  home  proc  root  sys  tmp  to1  usr  var

ls /root/mnt/container2
bin  dev  etc  home  proc  root  sys  tmp  to2  usr  var
```

```
// 查看 writeLayer 目录结构
tree writeLayer/
writeLayer/
├── container1
│   └── to1
└── container2
    └── to2
```

接下来，用 mydocker exec 命令进入到 container1 容器中。创建 /to1/test1.txt 文件，写入 "hello container1"（写入数据卷的操作）。创建 /to1-1/test1.txt 文件，写入 "hello container1,to1-1,test1"。

```
./mydocker exec container1 sh
{"level":"info","msg":"container pid 1441","time":"2017-01-07T18:36:56+08:00"}
{"level":"info","msg":"command sh","time":"2017-01-07T18:36:56+08:00"}

/ # echo -e "hello container1" >> /to1/test1.txt

/ # mkdir to1-1
/ # echo -e "hello container1,to1-1,test1" >> /to1-1/test1.txt
```

在另外一个 terminal 窗口中，查看宿主机上 writeLayer 目录的内容。多了 /container1/to1-1 目录和 /container1/to1-1/test1.txt 文件。

```
tree writeLayer/
writeLayer/
├── container1
│   ├── root
│   ├── to1
│   └── to1-1
│       └── test1.txt
└── container2
    └── to2

6 directories, 1 file
```

查看 /container1/to1-1/test1.txt 文件，符合在容器里的创建文件的内容。

```
root@iZ62wrfki2jZ:~# cat writeLayer/container1/to1-1/test1.txt
hello container1,to1-1,test1
```

下面查看写在数据卷中的文件。继续在该窗口查看 from1 文件目录的内容。该目录下有 test1.txt 文件，并且文件内容为“hello container1”。

```
ls from1
test1.txt
root@iZ62wrfki2jZ:~# cat from1/test1.txt
hello container1
```

接下来，验证一下 mydocker commit 命令。首先，退出 container1 容器，然后执行 ./mydocker commit container1 image1 命令，查看 /root 目录的内容，image1.tar 的镜像制作完成。

```
./mydocker commit container1 image1
/root/image1.tar

ls /root
busybox  busybox.tar  from1  from2  image1.tar  mnt  writeLayer
```

接下来，停止 container1 容器并删除。

```
./mydocker stop container1
./mydocker rm container1
./mydocker ps
ID           NAME          PID         STATUS      COMMAND     CREATED
1735876290   container2    1480        running     top         2017-01-07 18:16:56
```

查看 /root 目录下的内容，可以看到，即使删除了容器，宿主机上的数据卷目录并没有被删除，内容也保持不变。

```
root@iZ62wrfki2jZ:~# ls /root
busybox  busybox.tar  from1  from2  image1.tar  mnt  writeLayer
root@iZ62wrfki2jZ:~# ls from1
test1.txt
root@iZ62wrfki2jZ:~# cat from1/test1.txt
hello container1
```

container1 容器的根文件目录和可读写层被删除。

```
root@iZ62wrfki2jZ:~# tree writeLayer/
writeLayer/
└── container2
    └── to2

2 directories, 0 files
root@iZ62wrfki2jZ:~# ls mnt
container2
```

最后，用以上制作的 image1.tar 镜像来创建 container3，仍然把宿主机 /root/from1 挂载到容器 /to1 目录下。

```
./mydocker run -d --name container3 -v /root/from1:/to1 image1 top
{"level":"info","msg":"createTty false","time":"2017-01-07T19:03:44+08:00"}
{"level":"info","msg":"Mkdir parent dir /root/from1 error. mkdir /root/from1:
file exists","time":"2017-01-07T19:03:45+08:00"}
{"level":"info","msg":"NewWorkSpace volume urls [\"/root/from1\" \"/
to1\"]","time":"2017-01-07T19:03:45+08:00"}
{"level":"info","msg":"command all is top","time":"2017-01-07T19:03:45+08:00"}
{"level":"warning","msg":"remove cgroup fail remove /sys/fs/cgroup/
memory/mydocker-cgroup/memory.kmem.tcp.max_usage_in_bytes: operation not
permitted","time":"2017-01-07T19:03:45+08:00"}
{"level":"warning","msg":"remove cgroup fail remove /sys/fs/cgroup/cpu/mydocker-
cgroup/cpu.stat: operation not permitted","time":"2017-01-07T19:03:45+08:00"}

./mydocker ps
ID           NAME         PID         STATUS      COMMAND     CREATED
1735876290   container2   1480        running     top         2017-01-07 18:16:56
2348758530   container3   1985        running     top         2017-01-07 19:03:45
```

在另外一个 terminal 窗口中检查 /root 目录内容，目录下多了 image1。

```
ls /root
busybox  busybox.tar  from1  from2  image1  image1.tar  mnt  writeLayer
```

用 image1.tar 镜像启动容器 container3，进入容器 container3，可以发现写着“hello container1”的 /to1/test1.txt 文件。至此，说明我们成功地把容器 container1 数据卷 to1 的信息，重新写入了容器 container3 的数据卷 to1。

```
./mydocker exec container3 sh
{"level":"info","msg":"container pid 1713","time":"2017-01-08T00:40:04+08:00"}
{"level":"info","msg":"command sh","time":"2017-01-08T00:40:04+08:00"}
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   to1   usr   var
/ # cat /to1/test1.txt
hello container1
```

在容器 container3 容器内创建 /to1/test2.txt 文件，写入“hello container3”。

```
echo -e "hello container3" >> /to1/test2.txt
# cat /to1/test2.txt
hello container3
```

在另外一个终端窗口中检查宿主机 from1 目录内容，from1 目录下多了 test2.txt 文件，内容写有“hello container3”。说明数据卷挂载正常。

```
root@iZ62wrfki2jZ:~# ls from1/
```

```
test1.txt  test2.txt
root@iZ62wrfki2jZ:~# cat /root/from1/test2.txt
hello container3
```

5.8　实现容器指定环境变量运行

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-5.8
```

在 5.7 节中，实现了启动容器时使用管道将容器需要执行的命令传入容器中，而且对于文件类型的资源，可以通过 volume 挂载到容器中访问。那么环境变量呢？本节就来实现通过在启动容器时指定环境变量，让容器内运行的程序可以使用外部传递的环境变量。

5.8.1　修改 runCommand

在原来的基础上，增加 -e 选项，允许用户指定环境变量。这里由于环境变量可能是多个的，因此允许用户通过多次使用 -e 选项来传递多个环境变量。首先增加 -e 选项。

```
cli.StringSliceFlag{
            Name: "e",
            Usage: "set environment",
},
```

注意到这里的类型是 cli.StringSliceFlag，即字符串数组参数，因为这是针对传入多个环境变量的情况。对于参数的解析，需要增加对环境变量的解析，并且传递给 Run 函数。

```
envSlice := context.StringSlice("e")
Run(createTty, cmdArray, resConf, containerName, volume, imageName, envSlice)
```

这里，得到了用户传递过来的环境变量数组，然后传递给 Run 函数，用来启动容器。

5.8.2　修改 Run 函数

以下为修改 Run 函数的代码。

```
func Run(tty bool, comArray []string, res *subsystems.ResourceConfig,
containerName, volume, imageName string, envSlice []string) {
    containerID := randStringBytes(10)
    if containerName == "" {
        containerName = containerID
```

```
    }

    // 将环境变量传递给 process
    parent, writePipe := container.NewParentProcess(tty, containerName, volume,
    imageName, envSlice)
    ........
}
```

5.8.3 修改 NewParentProcess 函数

```
......
cmd.ExtraFiles = []*os.File{readPipe}
cmd.Env = append(os.Environ(), envSlice...)
NewWorkSpace(volume, imageName, containerName)
......
```

这里直接使用 Go 提供的 command。由于原来的 command 实际就是容器启动的进程，所以只需要在原来的基础上，增加一下环境变量的配置即可。os.Environ() 的环境变量就是系统默认的配置。默认情况下，新启动进程的环境变量都是继承于原来父进程的环境变量，但是如果手动指定了环境变量，那么这里就会覆盖掉原来继承自父进程的变量。由于在容器的进程中，有时候还需要使用原来父进程的环境变量，比如 PATH 等，因此这里会使用 os.Environ() 来获取宿主机的环境变量，然后把自定义的变量加进去。

好了，这个增加环境变量的功能就完成了，下面来实验一下。

```
## 使用 -e 命令指定多个环境变量
[mydocker] ./mydocker run -ti --name bird -e bird=123 -e luck=bird busybox sh
{"level":"info","msg":"createTty true","time":"2017-01-07T09:17:49Z"}
{"level":"info","msg":"command all is sh","time":"2017-01-07T09:17:49Z"}
{"level":"info","msg":"init come on","time":"2017-01-07T09:17:49Z"}
{"level":"info","msg":"Current location is /root/mnt/bird","time":"2017-01-
07T09:17:49Z"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-07T09:17:49Z"}
## 查看 env
/ # env | grep bird
luck=bird
bird=123
/ #
```

这里可以看到，手动指定的环境变量 bird=123、luck=bird 都已经可以在容器内可见了。下面创建一个后台运行的容器，查看一下是否可以。

```
## 以 -d 命令启动一个后台运行的容器并且指定环境变量
```

```
[mydocker]./mydocker run -d --name bird -e bird=123 -e luck=bird busybox top
{"level":"info","msg":"createTty false","time":"2017-01-07T09:20:12Z"}
{"level":"info","msg":"command all is top","time":"2017-01-07T09:20:12Z"}


## 查看容器运行的状态
root@ubuntu:[mydocker]# ./mydocker ps
ID           NAME        PID         STATUS      COMMAND     CREATED
6408899776   bird        14902       running     top         2017-01-07 09:20:12

## 使用 exec 进入容器中
root@[ubuntu]:[mydocker]# ./mydocker exec bird sh
{"level":"info","msg":"container pid 14902","time":"2017-01-07T09:20:22Z"}
{"level":"info","msg":"command sh","time":"2017-01-07T09:20:22Z"}

## 查看容器内进程
/ # ps -ef
PID   USER     TIME   COMMAND
    1 root       0:00 top
    4 root       0:00 sh -c sh
    5 root       0:00 sh
    7 root       0:00 ps -ef

## 查看容器内环境变量，发现并没有我们设置的环境变量
/ # env | grep bird
/ #
```

这里不能使用 env 命令获取设置环境变量的原因是，因为 exec 命令其实是 mydocker 发起的另外一个进程，这个进程的父进程其实是宿主机的，并不是容器内的。因为在 Cgo 里面使用了 setns 系统调用，才使得这个进程进入到了容器内的命名空间，但是由于环境变量是继承自父进程的，因此这个 exec 进程的环境变量其实是继承自宿主机的，所以在 exec 进程内看到的环境变量其实是宿主机的环境变量。但是，只要是容器内 PID 为 1 的进程，创建出来的进程都会继承它的环境变量。下面修改 exec 命令来直接使用 env 命令查看容器内环境变量的功能。

5.8.4　修改 mydocker exec 命令

首先提供了一个函数，可以根据指定的 PID 来获取对应进程的环境变量。

```
func getEnvsByPid(pid string) []string {
    // 进程环境变量存放的位置是 /proc/PID/environ
    path := fmt.Sprintf("/proc/%s/environ", pid)
    contentBytes, err := ioutil.ReadFile(path)
```

```
    if err != nil {
        log.Errorf("Read file %s error %v", path, err)
        return nil
    }
    // 多个环境变量中的分隔符是 \u0000
    envs := strings.Split(string(contentBytes), "\u0000")
    return envs
}
```

由于进程存放环境变量的位置是 /proc/<PID>/environ，因此根据给定的 PID 去读取这个文件，便可以获取环境变量。在文件的内容中，每个环境变量之间是通过 \u0000 分割的，因此以此为标记来获取环境变量数组。

```
func ExecContainer(containerName string, comArray []string) {
    pid, err := GetContainerPidByName(containerName)
    if err != nil {
       log.Errorf("Exec container getContainerPidByName %s error %v", containerName,
err)
        return
    }

    cmdStr := strings.Join(comArray, " ")
    log.Infof("container pid %s", pid)
    log.Infof("command %s", cmdStr)

    cmd := exec.Command("/proc/self/exe", "exec")
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    os.Setenv(ENV_EXEC_PID, pid)
    os.Setenv(ENV_EXEC_CMD, cmdStr)
    // 获取对应的 PID 环境变量，其实也就是容器的环境变量
    containerEnvs := getEnvsByPid(pid)
    // 将宿主机的环境变量和容器的环境变量都放置到 exec 进程内
    cmd.Env = append(os.Environ(), containerEnvs...)

    if err := cmd.Run(); err != nil {
        log.Errorf("Exec container %s error %v", containerName, err)
    }
}
```

这里由于 exec 命令依然需要宿主机的一些环境变量，因此将宿主机的环境变量和容器的环

境变量都一起放置到 exec 的进程内，这样就可以获取到容器内的环境变量了。下面来试一下。

```
## 创建后台运行进程
[mydocker]./mydocker run -d --name bird -e bird=123 -e luck=bird busybox top
{"level":"info","msg":"createTty false","time":"2017-01-07T10:48:28Z"}
{"level":"info","msg":"command all is top","time":"2017-01-07T10:48:28Z"}
root@ubuntu:[mydocker]# ./mydocker ps
ID           NAME         PID          STATUS       COMMAND      CREATED
8003476429   bird         16383        running      top          2017-01-07 10:48:28

## 进入到容器中
root@ubuntu:[mydocker]# ./mydocker exec bird sh
{"level":"info","msg":"container pid 16383","time":"2017-01-07T10:48:35Z"}
{"level":"info","msg":"command sh","time":"2017-01-07T10:48:35Z"}

## 查看容器内环境变量
/ # env | grep bird
luck=bird
bird=123
/ #
```

这里可以发现，mydocker exec 的进程已经可以获取到前面的 mydocker run 时设置的环境变量了。

5.9　小结

本章实现了容器操作的基本功能。首先实现了容器的后台运行，然后将容器的状态在文件系统上做了存储。通过这些存储信息，又可以实现列出当前容器信息的功能。 并且，基于后台运行的容器，我们可以去手动停止容器，并清除掉容器的存储信息。最后修改了上一章镜像的存储结构，使得多个容器可以并存，且存储的内容互不干扰。

第6章 容器网络

在第4章和第5章中，通过Namespace和Cgroups技术实现了容器进程的隔离，并且通过AUFS让容器拥有自己的“文件系统”，容器中的进程所感受到的环境就像在一台虚拟机上一样，但是这台“虚拟机”还没有插上“网线”，那么这一章就会通过网络虚拟化技术构建容器的网络，给这个“虚拟机”插上网线。

6.1 网络虚拟化技术介绍

在第2章“基础技术”中，介绍了Net Namespace。在那个实例中，通过Namespace的技术给容器配置了独立的网络空间。那么，要怎么给这个网络空间增加网络的配置呢？这就需要先了解一下下面这些网络虚拟化技术。

6.1.1 Linux虚拟网络设备

我们都知道，Linux实际是通过网络设备去操作和使用网卡的，系统装了一个网卡之后会为其生成一个网络设备实例，比如eth0。而随着网络虚拟化技术的发展，Linux支持创建出虚拟化的设备，可以通过虚拟化设备的组合实现多种多样的功能和网络拓扑。常见的虚拟化设备有Veth、 Bridge、802.1.q VLAN device、TAP，这里主要介绍构建容器网络要用到的Veth和Bridge。

Linux Veth

Veth 是成对出现的虚拟网络设备，发送到 Veth 一端虚拟设备的请求会从另一端的虚拟设备中发出。在容器的虚拟化场景中，经常会使用 Veth 连接不同的网络 Namespace，如下。

```
[/]$ # 创建两个网络 Namespace
[/]$ sudo ip netns add ns1
[/]$ sudo ip netns add ns2
[/]$ # 创建一对 Veth
[/]$ sudo ip link add veth0 type veth peer name veth1
[/]$ # 分别将两个 Veth 移到两个 Namespace 中
[/]$ sudo ip link set veth0 netns ns1
[/]$ sudo ip link set veth1 netns ns2
[/]$ # 去 ns1 的 namespace 中查看网络设备
[/]$ sudo ip netns exec ns1 ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
17: veth0@if16: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
group default qlen 1000
    link/ether 8e:bb:18:8a:c2:85 brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

如图 6.1，在 ns1 和 ns2 的 Namespace 中，除 loopback 的设备以外就只看到了一个网络设备。当请求发送到这个虚拟网络设备时，都会原封不动地从另外一个网络 Namespace 的网络接口中出来。例如，给两端分别配置不同的地址后，向虚拟网络设备的一端发送请求，就能到达这个虚拟网络设备对应的另一端。

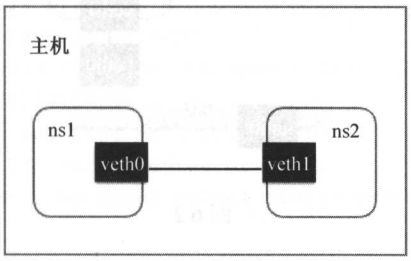


图 6.1

```
[/]$ # 配置每个 veth 的网络地址和 Namespace 的路由
[/]$ sudo ip netns exec ns1 ifconfig veth0 172.18.0.2/24 up
[/]$ sudo ip netns exec ns2 ifconfig veth1 172.18.0.3/24 up
[/]$ sudo ip netns exec ns1 route add default dev veth0
[/]$ sudo ip netns exec ns2 route add default dev veth1
[/]$ # 通过 veth 一端出去的包，另外一端能够直接接收到
[/]$ sudo ip netns exec ns1 ping -c 1 172.18.0.3
```

```
PING 172.18.0.3 (172.18.0.3) 56(84) bytes of data.
64 bytes from 172.18.0.3: icmp_seq=1 ttl=64 time=0.216 ms

--- 172.18.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.216/0.216/0.216/0.000 ms
```

Linux Bridge

Bridge 虚拟设备是用来桥接的网络设备，它相当于现实世界中的交换机，可以连接不同的网络设备，当请求到达 Bridge 设备时，可以通过报文中的 Mac 地址进行广播或转发。例如，创建一个 Bridge 设备，来连接 Namespace 中的网络设备和宿主机上的网络，如图 6.2 所示。

```
[/]$ # 创建 Veth 设备并将一端移入 Namespace
[/]$ sudo ip netns add ns1
[/]$ sudo ip link add veth0 type veth peer name veth1
[/]$ sudo ip link veth1 setns ns1
[/]$ # 创建网桥
[/]$ sudo brctl addbr br0
[/]$ # 挂载网络设备
[/]$ sudo brctl addif br0 eth0
[/]$ sudo brctl addif br0 veth0
```

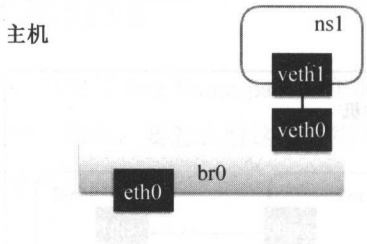


图 6.2

6.1.2 Linux 路由表

路由表是 Linux 内核的一个模块，通过定义路由表来决定在某个网络 Namespace 中包的流向，从而定义请求会到哪个网络设备上。继续用上面的例子来演示一下路由表的功能，代码如下，图示如图 6.3。

```
[/]$ # 启动虚拟网络设备，并设置它在 Net Namespace 中的 IP 地址
[/]$ sudo ip link set veth0 up
[/]$ sudo ip link set br0 up
[/]$ sudo ip netns exec ns1 ifconfig veth1 172.18.0.2/24 up
```

```
[/]$ # 分别设置 ns1 网络空间的路由和宿主机上的路由
[/]$ #default 代表 0.0.0.0/0，即在 Net Namespace 中所有流量都经过 veth1 的网络设备流出
[/]$ sudo ip netns exec ns1 route add default dev veth1
[/]$ # 在宿主机上将 172.18.0.0/24 的网段请求路由到 br0 的网桥
[/]$ sudo route add -net 172.18.0.0/24 dev br0
```

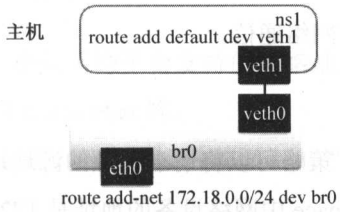


图 6.3

通过设置路由，对 IP 地址的请求就能正确被路由到对应的网络设备上，从而实现通信，如下。

```
[~]$ # 查看宿主机的 IP 地址
[~]$ sudo ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 08:00:27:0e:94:e9
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe0e:94e9/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4521 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1028 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4959937 (4.9 MB)  TX bytes:70270 (70.2 KB)
[/]$ # 从 Namespace 中访问宿主机的地址
[/]$ sudo ip netns exec ns1 ping -c 1 10.0.2.15
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.089 ms

--- 10.0.2.15 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.089/0.089/0.089/0.000 ms
[/]$ # 从宿主机访问 Namespace 中的网络地址
[/]$ ping -c 1 172.18.0.2
PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
64 bytes from 172.18.0.2: icmp_seq=1 ttl=64 time=0.047 ms

--- 172.18.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.047/0.047/0.047/0.000 ms
```

6.1.3 Linux iptables

iptables 是对 Linux 内核的 netfilter 模块进行操作和展示的工具，用来管理包的流动和转送。iptables 定义了一套链式处理的结构，在网络包传输的各个阶段可以使用不同的策略对包进行加工、传送或丢弃。在容器虚拟化的技术中，经常会用到两种策略 MASQUERADE 和 DNAT，用于容器和宿主机外部的网络通信。

MASQUERADE

iptables 中的 MASQUERADE 策略可以将请求包中的源地址转换成一个网络设备的地址，比如 6.1.2 小节介绍的那个 Namespace 中网络设备的地址是 172.18.0.2，这个地址虽然在宿主机上可以路由到 br0 的网桥，但是到达宿主机外部之后，是不知道如何路由到这个 IP 地址的，所以如果请求外部地址的话，需要先通过 MASQUERADE 策略将这个 IP 转换成宿主机出口网卡的 IP，如下。

```
[/]$ # 打开 IP 转发
[/]$ sudo sysctl -w net.ipv4.conf.all.forwarding=1
net.ipv4.conf.all.forwarding = 1
[/]$ # 对 Namespace 中发出的包添加网络地址转换
[/]$ sudo iptables -t nat -A POSTROUTING -s 172.18.0.0/24 -o eth0 -j MASQUERADE
```

在 Namespace 中请求宿主机外部地址时，将 Namespace 中的源地址转换成宿主机的地址作为源地址，就可以在 Namespace 中访问宿主机外的网络了。

DNAT

iptables 中的 DNAT 策略也是做网络地址的转换，不过它是要更换目标地址，经常用于将内部网络地址的端口映射到外部去。比如，上面那个例子中的 Namespace 如果需要提供服务给宿主机之外的应用去请求要怎么办呢？外部应用没办法直接路由到 172.18.0.2 这个地址，这时候就可以用到 DNAT 策略。

```
[/]$ # 将到宿主机上 80 端口的请求转发到 Namespace 的 IP 上
[/]$ sudo iptables -t nat -A PREROUTING -p tcp -m tcp --dport 80 -j DNAT --to-
destination 172.18.0.2:80
```

这样就可以把宿主机上 80 端口的 TCP 请求转发到 Namespace 中的地址 172.18.0.2:80，从而实现外部的应用调用。

6.1.4　Go 语言网络库介绍

6.1.3 小节中通过命令手动地配置了容器的网络，那么用 Go 语言的话又怎么调用配置呢？下面就介绍一下 6.1.5 小节中会用到的几个 Go 语言的网络操作库。

net 库

net 库是 Go 语言内置的库，提供了跨平台支持的网络地址处理，以及各种常见协议的 IO 支持，比如 TCP、UDP、DNS、Unix Socket 等。

它的引用方法很简单。在 Go 语言的源码文件前面引入“net”的包，就可以使用包中的数据结构和函数。

```
import "net"
```

后面会主要用到这个包中所定义的网络地址的数据结构和对网络地址的处理。

- net.IP：这个类型定义了 IP 地址的数据结构，并可以通过 ParseIP 和 String 方法将字符串与其转换。
- net.IPNet：这个类型定义了 IP 段的数据结构，比如 192.168.0.0/16 这样的网段，同样可以通过 ParseCIDR 和 String 方法与字符串转换。

github.com/vishvananda/netlink 库

github.com/vishvananda/netlink 是 Go 语言的操作网络接口、路由表等配置的库，使用它的调用相当于我们通过 IP 命令去管理网络接口。

在 Go 语言源码文件中，引入 github.com/vishvananda/netlink 库之后，就可以通过 netlink 操作机器上的网络配置。

```
import "github.com/vishvananda/netlink"

        ...
    netlink.LinkAdd(xxxlink)
    netlink.LinkDel(xxxlink)
        ...
```

在这个库中包含了很多用于容器网络接口配置的数据结构和函数，后面在使用到它的具体数据结构和函数时会具体介绍它们的功能。

github.com/vishvananda/netns 库

6.7.1 小节中通过 ip netns exec 进入到容器的 Net Namespace 中去执行网络的配置，github.

com/vishvananda/netns 就是 Go 语言版进出 Net Namespace 的库，通过这个库，可以让 netlink 库中配置网络接口的代码在某个容器的 Net Namespace 中执行。

```
import "github.com/vishvananda/netns"

        ...
    netns.Set(continer_netns)
        ...
```

可以通过调用 netns 的 Set 方法将当前代码执行的线程加入到指定的 Net Namespace 中，在 6.2 节中使用到这个库的函数时会对其用法做具体的介绍。

6.2 构建容器网络模型

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-6.5
```

在上一节中，通过 Linux 的网络虚拟化技术构建了一个网络，连接了容器的网络 Namespace 和宿主机网络，其中做了命名空间创建、设备创建、地址分配、挂载设备和地址转换配置等操作，下面会将这些操作抽象出网络的模型以便于用代码实现上面一系列流程。

6.2.1 模型

如图 6.4 所示，为整体的网络模型结构图，其中包含了整个模型的组件和流程。首先，需要抽象出容器网络的两个对象——网络和网络端点。

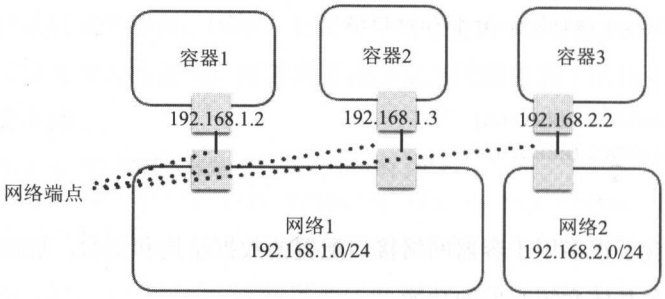


图 6.4

网络

网络是容器的一个集合，在这个网络上的容器可以通过这个网络互相通信，就像挂载到同

一个 Linux Bridge 设备上的网络设备一样，可以直接通过 Bridge 设备实现网络互连；连接到同一个网络中的容器也可以通过这个网络和网络中别的容器互连。网络中会包括这个网络相关的配置，比如网络的容器地址段、网络操作所调用的网络驱动等信息。

```
type Network struct {
    Name string // 网络名
    IpRange *net.IPNet // 地址段
    Driver string // 网络驱动名
}
```

网络端点

网络端点是用于连接容器与网络的，保证容器内部与网络的通信。像上一节中用到的 Veth 设备，一端挂载到容器内部，另一端挂载到 Bridge 上，就能保证容器和网络的通信。网络端点中会包括连接到网络的一些信息，比如地址、Veth 设备、端口映射、连接的容器和网络等信息。

```
type Endpoint struct {
    ID string `json:"id"`
    Device netlink.Veth `json:"dev"`
    IPAddress net.IP `json:"ip"`
    MacAddress net.HardwareAddr `json:"mac"`
    PortMapping []string `json:"portmapping"`
    Network     *Network
}
```

而网络端点的信息传输需要靠网络功能的两个组件配合完成，这两个组件分别为网络驱动和 IPAM，具体介绍如下。

网络驱动

网络驱动（Network Driver）是一个网络功能中的组件，不同的驱动对网络的创建、连接、销毁的策略不同，通过在创建网络时指定不同的网络驱动来定义使用哪个驱动做网络的配置。它的接口定义如下。

```
type NetworkDriver interface {
// 驱动名
Name() string
// 创建网络
Create(subnet string, name string) (*Network, error)
// 删除网络
Delete(network Network) error
// 连接容器网络端点到网络
```

```
Connect(network *Network, endpoint *Endpoint) error
// 从网络上移除容器网络端点
Disconnect(network Network, endpoint *Endpoint) error
}
```

6.3 节会以 Bridge 驱动为例子，介绍如何实现一个网络驱动。

IPAM

IPAM 也是网络功能中的一个组件，用于网络 IP 地址的分配和释放，包括容器的 IP 地址和网络网关的 IP 地址。下一节会具体介绍 IPAM 的实现，它的主要功能如下。

- IPAM.Allocate(subnet *net.IPNet) 从指定的 subnet 网段中分配 IP 地址。
- IPAM.Release(subnet net.IPNet, ipaddr net.IP) 从指定的 subnet 网段中释放掉指定的 IP 地址。

6.2.2 调用关系

创建网络

创建网络中将实现通过使用 network create 命令创建一个容器网络。

```
mydocker network create --subnet 192.168.0.0/24 --driver bridge testbridgenet
```

通过 Bridge 的网络驱动创建一个网络，网段是 192.168.0.0/24，网络驱动是 Bridge。

图 6.5 是创建网络的流程图。

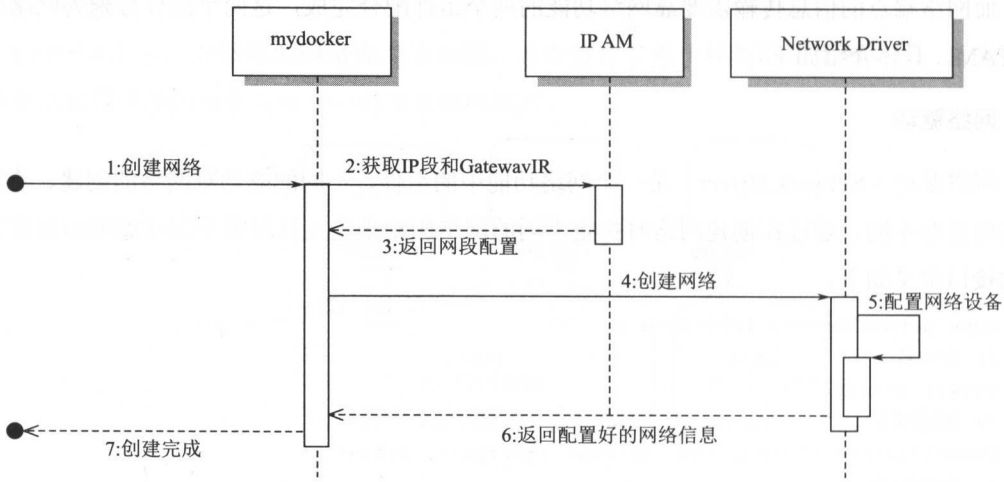


图 6.5

上图中的 IPAM 和 Network Driver 是两个组件，IPAM 负责通过传入的 IP 网段去分配一个可用的 IP 地址给容器和网络的网关，比如网络的网段是 192.168.0.0/16，那么通过 IPAM 获取这个网段的容器地址就是在这个网段中的一个 IP 地址，然后用于分配给容器的连接端点，保证网络中的容器 IP 不会冲突。而 Network Driver 是用于网络的管理的，例如在创建网络时完成网络初始化动作及在容器启动时完成网络端点配置，像 Bridge 的驱动对应的动作就是创建 Linux Bridge 和挂载 Veth 设备。后面几节会对 IPAM 和 Network Driver 的实现做详细介绍。

如图 6.5 所示，在调用命令创建网络时，先通过 IPAM 获取网络的网关 IP，然后再调用网络驱动去设置网络的信息，比如 Bridge 的驱动将会创建 Linux Bridge 网络设备和相应的 iptables 规则，最终将网络的信息返回给调用者。

```
// 创建网络
func CreateNetwork(driver, subnet, name string) error {
   // ParseCIDR 是 Golang net 包的函数，功能是将网段的字符串转换成 net.IPNet 的对象
    _, cidr, _ := net.ParseCIDR(subnet)
    // 通过 IPAM 分配网关 IP，获取到网段中第一个 IP 作为网关的 IP，下面几节会具体介绍它的实现
    gatewayIp, err := ipAllocator.Allocate(cidr)
    if err != nil {
        return err
    }
    cidr.IP = gatewayIp

    /* 调用指定的网络驱动创建网络，这里的 drivers 字典是各个网络驱动的实例字典，通过调用网络驱动
   的 Create 方法创建网络，后面会以 Bridge 驱动为例介绍它的实现 */
    nw, err := drivers[driver].Create(cidr.String(), name)
    if err != nil {
        return err
    }
    // 保存网络信息，将网络的信息保存在文件系统中，以便查询和在网络上连接网络端点
    return nw.dump(defaultNetworkPath)
}
```

其中，Network.dump 和 Network.load 方法是将这个网络的配置信息保存在文件系统中，或者从网络的配置目录中的文件读取到网络的配置，以便网络查询及在这个网络上连接网络端点。

```
func (nw *Network) dump(dumpPath string) error {
    // 检查保存的目录是否存在，不存在则创建
    if _, err := os.Stat(dumpPath); err != nil {
        if os.IsNotExist(err) {
            os.MkdirAll(dumpPath, 0644)
```

```
        } else {
            return err
        }
    }
    // 保存的文件名是网络的名字
    nwPath := path.Join(dumpPath, nw.Name)
    // 打开保存的文件用于写入，后面打开的模式参数分别是存在内容则清空、只写入、不存在则创建
    nwFile, err := os.OpenFile(nwPath, os.O_TRUNC | os.O_WRONLY | os.O_
CREATE, 0644)
    if err != nil {
        logrus.Errorf("error: ", err)
        return err
    }
    defer nwFile.Close()

    // 通过 json 的库序列化网络对象到 json 的字符串
    nwJson, err := json.Marshal(nw)
    if err != nil {
        logrus.Errorf("error: ", err)
        return err
    }

    // 将网络配置的 json 字符串写入到文件中
    _, err = nwFile.Write(nwJson)
    if err != nil {
        logrus.Errorf("error: ", err)
        return err
    }
    return nil
}

func (nw *Network) load(dumpPath string) error {
    // 打开配置文件
    nwConfigFile, err := os.Open(dumpPath)
    defer nwConfigFile.Close()
    if err != nil {
        return err
    }
    // 从配置文件中读取网络的配置 json 字符串
    nwJson := make([]byte, 2000)
    n, err := nwConfigFile.Read(nwJson)
    if err != nil {
        return err
    }
```

```
    // 通过 json 字符串反序列化出网络
    err = json.Unmarshal(nwJson[:n], nw)
    if err != nil {
        logrus.Errorf("Error load nw info", err)
        return err
    }
    return nil
}
```

创建容器并连接网络

通过创建容器时指定 --net 的参数，指定容器启动时连接的网络。

```
mydocker run -ti -p 80:80 --net testbridgenet xxxx
```

创建出的容器便可以通过 testbridgenet 这个网络与网络中的其他容器进行通信。

如图 6.6 所示，在调用创建容器时指定网络，首先会调用 IPAM，通过网络中定义的网段找到未分配的 IP 给容器，然后创建容器的网络端点，并调用这个网络的网络驱动连接网络与网络端点，最终完成网络端点的连接和配置。比如在 Bridge 驱动中就会将 Veth 设备挂载到 Linux Bridge 的网桥上。

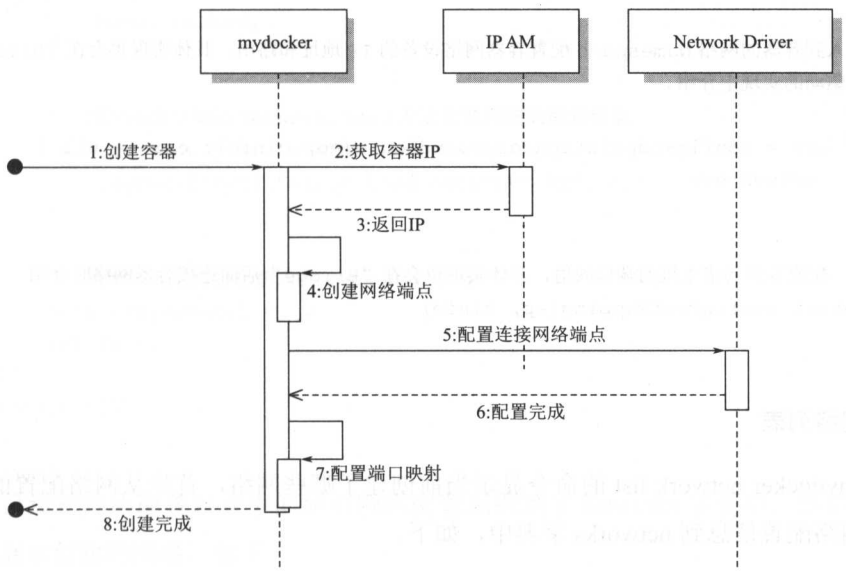


图 6.6

```
func Connect(networkName string, cinfo *container.ContainerInfo) error {
    // 从 networks 字典中取到容器连接的网络的信息，networks 字典中保存了当前已经创建的网络
    network, ok := networks[networkName]
```

```
    if !ok {
        return fmt.Errorf("No Such Network: %s", networkName)
    }
    // 通过调用 IPAM 从网络的网段中获取可用的 IP 作为容器 IP 地址
    ip, err := ipAllocator.Allocate(network.IpRange)
    if err != nil {
        return err
    }
    // 创建网络端点
    ep := &Endpoint{
        ID: fmt.Sprintf("%s-%s", cinfo.Id, networkName),
        IPAddress: ip,
        Network: network,
        PortMapping: cinfo.PortMapping,
    }
    /*
    调用网络驱动的“Connect”方法去连接和配置网络端点，后面会以“Bridge”网络驱动为例介绍
    一下它的实现。
    */
    if err = drivers[network.Driver].Connect(network, ep); err != nil {
        return err
    }
    /*
    进入到容器的网络 Namespace 配置容器网络设备的 IP 地址和路由，具体实现也会在“Bridge”网
    络驱动的实现中介绍。
    */
    if err = configEndpointIpAddressAndRoute(ep, cinfo); err != nil {
        return err
    }

    // 配置容器到宿主机的端口映射，具体实现也会在“Bridge”后面连接容器网络时介绍
    return configPortMapping(ep, cinfo)
}
```

展示网络列表

通过 mydocker network list 的命令显示当前创建了哪些网络，首先从网络配置的目录中加载所有的网络配置信息到 networks 字典中，如下。

```
func Init() error {
    // 加载网络驱动
    var bridgeDriver = BridgeNetworkDriver{}
    drivers[bridgeDriver.Name()] = &bridgeDriver
```

```
    // 判断网络的配置目录是否存在，不存在则创建
    if _, err := os.Stat(defaultNetworkPath); err != nil {
        if os.IsNotExist(err) {
            os.MkdirAll(defaultNetworkPath, 0644)
        } else {
            return err
        }
    }

    // 检查网络配置目录中的所有文件
    // filepath.Walk(path, func(string,os.FileInfo,error)) 函数会遍历指定的 path 目录
    // 并执行第二个参数中的函数指针去处理目录下的每一个文件
     filepath.Walk(defaultNetworkPath, func(nwPath string, info os.FileInfo, err
error) error {
        // 如果是目录则跳过
        if info.IsDir() {
            return nil
        }

        // 加载文件名作为网络名
        _, nwName := path.Split(nwPath)
        nw := &Network{
            Name: nwName,
        }

        // 调用前面介绍的 Network.load 方法加载网络的配置信息
        if err := nw.load(nwPath); err != nil {
            logrus.Errorf("error load network: %s", err)
        }

        // 将网络的配置信息加入到 networks 字典中
        networks[nwName] = nw
        return nil
    })
    return nil
}
```

上面已经把网络配置目录中的所有网络配置加载到了 networks 字典中，然后通过遍历这个字典来展示创建的网络，如下。

```
func ListNetwork() {
    // 通过前面在 mydocker ps 时介绍的 tabwriter 的库去展示网络
    w := tabwriter.NewWriter(os.Stdout, 12, 1, 3, ' ', 0)
    fmt.Fprint(w, "NAME\tIpRange\tDriver\n")
```

```
    // 遍历网络信息
    for _, nw := range networks {
        fmt.Fprintf(w, "%s\t%s\t%s\n",
            nw.Name,
            nw.IpRange.String(),
            nw.Driver,
        )
    }
    // 输出到标准输出
    if err := w.Flush(); err != nil {
        logrus.Errorf("Flush error %v", err)
        return
    }
}
```

删除网络

通过使用命令 mydocker network remove testbridgenet 删除已经创建的网络，调用的流程如图 6.7 所示。

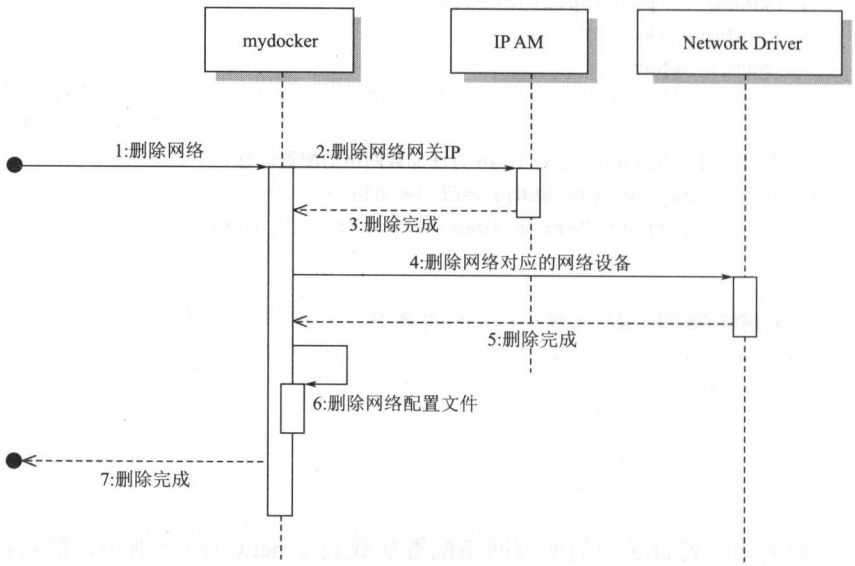


图 6.7

如图 6.7 所示，删除网络的时候，会先调用 IPAM 去释放网络所占用的网关 IP，然后再调用网络驱动去删除网络创建的一些设备与配置，最终从网络配置目录中删除网络对应的配置文件，如下。

```
func DeleteNetwork(networkName string) error {
    // 查找网络是否存在
    nw, ok := networks[networkName]
    if !ok {
        return fmt.Errorf("No Such Network: %s", networkName)
    }

    // 调用 IPAM 的实例 ipAllocator 释放网络网关的 IP
    if err := ipAllocator.Release(nw.IpRange, &nw.IpRange.IP); err != nil {
        return fmt.Errorf("Error Remove Network gateway ip: %s", err)
    }

    /* 调用网络驱动删除网络创建的设备与配置，后面会以 Bridge 驱动删除网络为例子介绍如何实现网络
    驱动删除网络。*/
    if err := drivers[nw.Driver].Delete(*nw); err != nil {
        return fmt.Errorf("Error Remove Network DriverError: %s", err)
    }

    // 从网络的配置目录中删除该网络对应的配置文件
    return nw.remove(defaultNetworkPath)
}
```

其中，Network.remove 会从网络配置目录中删除网络的配置文件。

```
func (nw *Network) remove(dumpPath string) error {
    // 网络对应的配置文件，即配置目录下的网络名文件
    // 检查文件状态，如果文件已经不存在就直接返回
    if _, err := os.Stat(path.Join(dumpPath, nw.Name)); err != nil {
        if os.IsNotExist(err) {
            return nil
        } else {
            return err
        }
    } else {
        // 调用 os.Remove 删除这个网络对应的配置文件
        return os.Remove(path.Join(dumpPath, nw.Name))
    }
}
```

6.3　容器地址分配

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
```

```
git checkout code-6.5
```

上一节介绍了实现容器网络的模型和流程，创建网络和连接容器的第一步就是分配容器的 IP 地址，分别用于网关 IP 和容器的网络端点 IP，以保证网络中 IP 地址的唯一。这一节会介绍怎样实现网络中 IP 地址的分配，即如何管理网段中 IP 地址的分配与释放。

6.3.1 bitmap 算法介绍

bitmap 算法，也叫位图算法，在大规模连续且少状态的数据处理中有很高的效率，比如要用到的 IP 地址分配。在网段中，某个 IP 地址有两种状态，1 表示已经被分配了，0 表示还未被分配，那么一个 IP 地址的状态就可以用一位来表示，并且通过这一位相对基础位的偏移也能够迅速定位到数据所在的位。例如，图 6.8 中的内存 IP 地址位的列表，通过地址相对于 192.168.0.0/24 的偏移找到所在的位，然后通过位中的值是 0 还是 1 去管理地址分配的信息。

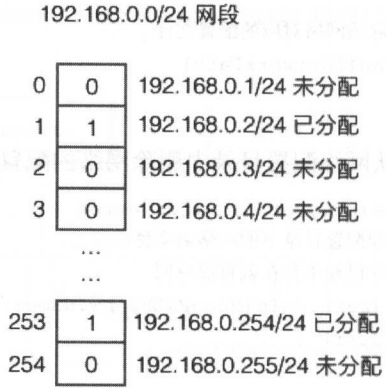


图 6.8

如图 6.8 所示，通过位图的方式实现 IP 地址的管理，在获取 IP 地址时，遍历这个数组，找到值为 0 的数组项的偏移，然后通过偏移和网段的配置计算出分配的 IP 地址，并将这个数组项置为 1，表明 IP 地址已经被分配。在释放的时候，原理也是一样的。

6.3.2 数据结构定义

下面，将通过代码来实现这一流程，首先介绍一下地址分配的数据结构的定义，如下。

```
const ipamDefaultAllocatorPath = "/var/run/mydocker/network/ipam/subnet.json"
// 存放 IP 地址分配信息
type IPAM struct {
```

```
    // 分配文件存放位置
    SubnetAllocatorPath string
    // 网段和位图算法的数组 map，key 是网段，value 是分配的位图数组
    Subnets *map[string]string
}
/* 初始化一个 IPAM 的对象，默认使用 "/var/run/mydocker/network/ipam/subnet.json" 作为分配信息存储位置。*/
var ipAllocator = &IPAM{
    SubnetAllocatorPath: ipamDefaultAllocatorPath,
}
```

注意：在这个定义中，为了代码实现简单和易于阅读，使用 string 中的一个字符表示一个状态位，实际上可以采用一位表示一个是否分配的状态位，这样资源会有更低的消耗。

通过将分配的信息序列化成 json 文件或将 json 文件以反序列化的方式保存和读取网段分配的信息，如下。

```
// 加载网段地址分配信息
func (ipam *IPAM) load() error {
    // 通过 os.Stat 函数检查存储文件状态，如果不存在，则说明之前没有分配，则不需要加载
    if _, err := os.Stat(ipam.SubnetAllocatorPath); err != nil {
        if os.IsNotExist(err) {
            return nil
        } else {
            return err
        }
    }
    // 打开并读取存储文件
    subnetConfigFile, err := os.Open(ipam.SubnetAllocatorPath)
    defer subnetConfigFile.Close()
    if err != nil {
        return err
    }
    subnetJson := make([]byte, 2000)
    n, err := subnetConfigFile.Read(subnetJson)
    if err != nil {
        return err
    }

    // 将文件中的内容反序列化出 IP 的分配信息
    err = json.Unmarshal(subnetJson[:n], ipam.Subnets)
    if err != nil {
        log.Errorf("Error dump allocation info, %v", err)
        return err
    }
```

```
    return nil
}
// 存储网段地址分配信息
func (ipam *IPAM) dump() error {
    // 检查存储文件所在文件夹是否存在，如果不存在则创建，path.Split 函数能够分隔目录和文件
    ipamConfigFileDir, _ := path.Split(ipam.SubnetAllocatorPath)
    if _, err := os.Stat(ipamConfigFileDir); err != nil {
        if os.IsNotExist(err) {
            // 创建文件夹，os.MkdirAll 相当于 mkdir -p <dir> 命令
            os.MkdirAll(ipamConfigFileDir, 0644)
        } else {
            return err
        }
    }
    // 打开存储文件，os.O_TRUNC 表示如果存在则清空，os.O_CREATE 表示如果不存在则创建
     subnetConfigFile, err := os.OpenFile(ipam.SubnetAllocatorPath, os.O_TRUNC |
os.O_WRONLY | os.O_CREATE, 0644)
    defer subnetConfigFile.Close()
    if err != nil {
        return err
    }
    // 序列化 ipam 对象到 json 串
    ipamConfigJson, err := json.Marshal(ipam.Subnets)
    if err != nil {
        return err
    }
    // 将序列化后的 json 串写入到配置文件中
    _, err = subnetConfigFile.Write(ipamConfigJson)
    if err != nil {
        return err
    }

    return nil
}
```

6.3.3 地址分配的实现

下面来介绍如何通过位图算法分配 IP 地址。IPAM.Allocate 函数用来实现在网段中分配一个可用的 IP 地址，并将 IP 地址分配信息记录到文件中，流程如下。

```
// 在网段中分配一个可用的 IP 地址
func (ipam *IPAM) Allocate(subnet *net.IPNet) (ip net.IP, err error) {
    // 存放网段中地址分配信息的数组
    ipam.Subnets = &map[string]string{}
```

```
// 从文件中加载已经分配的网段信息
err = ipam.load()
if err != nil {
    log.Errorf("Error load allocation info, %v", err)
}
//net.IPNet.Mask.Size() 函数会返回网段的子网掩码的总长度和网段前面的固定位的长度
// 比如“127.0.0.0/8”网段的子网掩码是“255.0.0.0”
// 那么 subnet.Mask.Size() 的返回值就是前面 255 所对应的位数和总位数，即 8 和 24
one, size := subnet.Mask.Size()

// 如果之前没有分配过这个网段，则初始化网段的分配配置
if _, exist := (*ipam.Subnets)[subnet.String()]; !exist {
    // 用“0”填满这个网段的配置，1 << uint8(size - one) 表示这个网段中有多少个可用地址
    //“size - one”是子网掩码后面的网络位数，2^(size - one) 表示网段中的可用 IP 数
    // 而 2^(size - one) 等价于 1 << uint8(size - one)
    (*ipam.Subnets)[subnet.String()] = strings.Repeat("0", 1 << uint8(size - one))
}

// 遍历网段的位图数组
for c := range((*ipam.Subnets)[subnet.String()]) {
    // 找到数组中为“0”的项和数组序号，即可以分配的 IP
    if (*ipam.Subnets)[subnet.String()][c] == '0' {
        // 设置这个为“0”的序号值为“1”，即分配这个 IP
        ipalloc := []byte((*ipam.Subnets)[subnet.String()])
        //Go 的字符串，创建之后就不能修改，所以通过转换成 byte 数组，修改后再转换成字符串赋值
        ipalloc[c] = '1'
        (*ipam.Subnets)[subnet.String()] = string(ipalloc)
        // 这里的 IP 为初始 IP，比如对于网段 192.168.0.0/16，这里就是 192.168.0.0
        ip = subnet.IP

        /*
        通过网段的 IP 与上面的偏移相加计算出分配的 IP 地址，由于 IP 地址是 uint 的一个数组，
        需要通过数组中的每一项加所需要的值，比如网段是 172.16.0.0/12，数组序号是 65555，
        那么在 [172,16,0,0] 上依次加 [uint8(65555 >> 24)、uint8(65555 >> 16)、
        uint8(65555 >> 8)、uint8(65555 >> 0)]，即 [0, 1, 0, 19]，那么获得的 IP 就
        是 172.17.0.19。
        */
        for t := uint(4); t > 0; t-=1 {
            []byte(ip)[4-t] += uint8(c >> ((t - 1) * 8))
        }
        // 由于此处 IP 是从 1 开始分配的，所以最后再加 1，最终得到分配的 IP 是 172.17.0.20
        ip[3]+=1
        break
    }
```

```
    }
    // 通过调用 dump() 将分配结果保存到文件中
    ipam.dump()
    return
}
```

6.3.4 地址释放的实现

下面介绍如何通过位图算法释放 IP 地址。IPAM.Release 函数用来实现将已分配的 IP 地址释放掉，参数是网段及要释放的 IP 地址，并将释放掉之后的网段分配信息保存到文件中。

```
func (ipam *IPAM) Release(subnet *net.IPNet, ipaddr *net.IP) error {
    ipam.Subnets = &map[string]string{}
    // 从文件中加载网段的分配信息
    err := ipam.load()
    if err != nil {
        log.Errorf("Error dump allocation info, %v", err)
    }

    // 计算 IP 地址在网段位图数组中的索引位置
    c := 0
    // 将 IP 地址转换成 4 个字节的表示方式
    releaseIP := ipaddr.To4()
    // 由于 IP 是从 1 开始分配的，所以转换成索引应减 1
    releaseIP[3]-=1
    for t := uint(4); t > 0; t-=1 {
        /* 与分配 IP 相反，释放 IP 获得索引的方式是 IP 地址的每一位相减之后分别左移将对应的数值加
        到索引上。*/
        c += int(releaseIP[t-1] - subnet.IP[t-1]) << ((4-t) * 8)
    }

    // 将分配的位图数组中索引位置的值置为 0
    ipalloc := []byte((*ipam.Subnets)[subnet.String()])
    ipalloc[c] = '0'
    (*ipam.Subnets)[subnet.String()] = string(ipalloc)

    // 保存释放掉 IP 之后的网段 IP 分配信息
    ipam.dump()
    return nil
}
```

6.3.5 测试

通过两个单元测试来测试网段中 IP 的分配和释放，如下。

```
func TestAllocate(t *testing.T) {
    // 在 192.168.0.0/24 网段中分配 IP
    _, ipnet, _ := net.ParseCIDR("192.168.0.0/24")
    ip, _ := ipAllocator.Allocate(ipnet)
    t.Logf("alloc ip: %v", ip)
}

func TestRelease(t *testing.T) {
    // 在 192.168.0.0/24 网段中释放刚才分配的 192.168.0.1 的 IP
    ip, ipnet, _ := net.ParseCIDR("192.168.0.1/24")
    ipAllocator.Release(ipnet, &ip)
}
```

如下为测试结果。

```
[/]$ # 得到分配的 IP 192.168.0.1
[/]$ godep go test -v -run "^TestAllocate$"
=== RUN   TestAllocate
--- PASS: TestAllocate (0.00s)
    ipam_test.go:11: alloc ip: 192.168.0.1
PASS
ok
github.com/xianlubird/mydocker/network  0.008s
[/]$ # 查看分配后的存储配置的文件，可以看到网段的第 1 位被置为了 1
[/]$ cat /var/run/mydocker/network/ipam/subnet.json
{"192.168.0.0/24":"1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000"}
[/]$ # 测试释放刚才分配的那个 IP
[/]$ godep go test -v -run "^TestRelease$"
=== RUN   TestRelease
--- PASS: TestRelease (0.00s)
PASS
ok
github.com/xianlubird/mydocker/network  0.009s
[/]$ # 查看释放掉 IP 之后存储配置的文件，可以看到 192.168.0.1 对应的第 1 位已经被重新置为 0
[/]$ cat /var/run/mydocker/network/ipam/subnet.json
{"192.168.0.0/24":"0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000"}
```

6.4 创建 Bridge 网络

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-6.5
```

在 6.3 节中，我们构建了容器。这一节将实现通过 mydocker network create -d bridge testnet 的方式创建和配置 Linux Bridge，供容器的网络端点挂载，即实现 NetworkDriver 接口的 Create 方法。

6.4.1 Bridge Driver Create 实现

前面介绍了 NetworkDriver 的接口定义，其中创建网络的方法是 Create，输入是网络的网段、网关及网络名，返回的是创建好的网络对象。

```
func (d *BridgeNetworkDriver) Create(subnet string, name string) (*Network,
error) {
    // 通过 net 包中的 net.ParseCIDR 方法，取到网段的字符串中的网关 IP 地址和网络 IP 段
    ip, ipRange, _ := net.ParseCIDR(subnet)
    ipRange.IP = ip
    // 初始化网络对象
    n := &Network {
        Name: name,
        IpRange: ipRange,
    }
    // 配置 Linux Bridge，6.4.2 小节会介绍这个函数做了些什么
    err := d.initBridge(n)
    if err != nil {
        log.Errorf("error init bridge: %v", err)
    }
    // 返回配置好的网络
    return n, err
}
```

6.4.2 Bridge Driver 初始化 Linux Bridge 流程

在本章的第一节中，曾手动配置了一个供 Veth 挂载和互相通信的 Linux Bridge，这一节会将此过程自动化起来，用来做容器之间及容器同外部的通信，流程如图 6.9 所示。

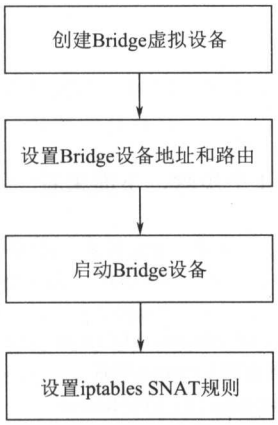


图 6.9

从图 6.9 中可以看到初始化 Linux Bridge 的 4 个流程。首先创建 Bridge 虚拟设备，再设置 Bridge 设备的地址和路由，然后启动 Bridge 设备，最后设置 iptabels 的 SNAT 规则，保证挂载到这个 Bridge 上的容器的 Veth 能够访问外部网络。

```
// 初始化 Bridge 设备
func (d *BridgeNetworkDriver) initBridge(n *Network) error {
    // 1. 创建 Bridge 虚拟设备
    bridgeName := n.Name
    if err := createBridgeInterface(bridgeName); err != nil {
        return fmt.Errorf("Error add bridge:  %s, Error: %v", bridgeName, err)
    }

    // 2. 设置 Bridge 设备的地址和路由
    gatewayIP := *n.IpRange
    gatewayIP.IP = n.IpRange.IP
    if err := setInterfaceIP(bridgeName, gatewayIP.String()); err != nil {
        return fmt.Errorf("Error assigning address: %s on bridge: %s with an
error of: %v", gatewayIP, bridgeName, err)
    }

    // 3. 启动 Bridge 设备
    if err := setInterfaceUP(bridgeName); err != nil {
        return fmt.Errorf("Error set bridge up: %s, Error: %v", bridgeName, err)
    }

    // 4. 设置 iptabels 的 SNAT 规则
    if err := setupIPTables(bridgeName, n.IpRange); err != nil {
        return fmt.Errorf("Error setting iptables for %s: %v", bridgeName, err)
```

```
    }

    return nil
}
```

以上就是初始化 Bridge 设备的 4 个步骤，下面来看一下每个步骤的实现。

创建 Bridge 虚拟设备

```
// 创建 Linux Bridge 设备
func createBridgeInterface(bridgeName string) error {
    // 先检查是否已经存在了这个同名的 Bridge 设备
    _, err := net.InterfaceByName(bridgeName)
    // 如果已经存在或者报错则返回创建错误
    if err == nil || !strings.Contains(err.Error(), "no such network interface") {
        return err
    }

    // 初始化一个 netlink 的 Link 基础对象，Link 的名字即 Bridge 虚拟设备的名字
    la := netlink.NewLinkAttrs()
    la.Name = bridgeName

    // 使用刚才创建的 Link 的属性创建 netlink 的 Bridge 对象
    br := &netlink.Bridge{la}
    // 调用 netlink 的 Linkadd 方法，创建 Bridge 虚拟网络设备
    // netlink 的 Linkadd 方法是用来创建虚拟网络设备的，相当于 ip link add xxxx
    if err := netlink.LinkAdd(br); err != nil {
        return fmt.Errorf("Bridge creation failed for bridge %s: %v", bridgeName, err)
    }
    return nil
}
```

通过 netlink 的 LinkAdd 方法，创建出了 Linux Bridge 的虚拟设备。

设置 Bridge 设备的地址和路由

```
// 设置一个网络接口的 IP 地址，例如 setInterfaceIP("testbridge", "192.168.0.1/24")
func setInterfaceIP(name string, rawIP string) error {
    // 通过 netlink 的 LinkByName 方法找到需要设置的网络接口
    iface, err := netlink.LinkByName(name)
    if err != nil {
        return fmt.Errorf("error get interface: %v", err)
    }
    /*
    由于 netlink.ParseIPNet 是对 net.ParseCIDR 的一个封装，因此可以将 net.ParseCIDR 的返
```

```
    回值中的 IP 和 net 整合。
    */
    // 返回值中的 ipNet 既包含了网段的信息，192.168.0.0/24，也包含了原始的 ip 192.168.0.1
    ipNet, err := netlink.ParseIPNet(rawIP)
    if err != nil {
        return err
    }
    // 通过 netlink.AddrAdd 给网络接口配置地址，相当于 ip addr add xxx 的命令
    // 同时如果配置了地址所在网段的信息，例如 192.168.0.0/24
    // 还会配置路由表 192.168.0.0/24 转发到这个 testbridge 的网络接口上
    addr := &netlink.Addr{ipNet, "", 0, 0}
    return netlink.AddrAdd(iface, addr)
}
```

通过调用 netlink 的 AddrAdd 方法，配置 Linux Bridge 的地址和路由表。

启动 Bridge 设备

```
// 设置网络接口为 UP 状态
func setInterfaceUP(interfaceName string) error {
    iface, err := netlink.LinkByName(interfaceName)
    if err != nil {
        return fmt.Errorf("Error retrieving a link named [ %s ]: %v", iface.
Attrs().Name, err)
    }

    // 通过 “netlink” 的 “LinkSetUp” 方法设置接口状态为 “UP” 状态
    // 等价于 ip link set xxx up 命令
    if err := netlink.LinkSetUp(iface); err != nil {
        return fmt.Errorf("Error enabling interface for %s: %v", interfaceName, err)
    }
    return nil
}
```

Linux 的网络设备只有设置成 UP 状态后才能处理和转发请求。通过 netlink 的 LinkSetUp 方法，将创建的 Linux Bridge 设置为 UP 状态。

设置 iptabels Linux Bridge SNAT 规则

```
// 设置 iptables 对应 bridge 的 MASQUERADE 规则
func setupIPTables(bridgeName string, subnet *net.IPNet) error {
    // 由于 Go 语言没有直接操控 iptables 操作的库，所以需要通过命令的方式来配置
    // 创建 iptables 的命令
    // iptables -t nat -A POSTROUTING -s <bridgeName> ! -o <bridgeName> -j MASQUERADE
```

```
    iptablesCmd := fmt.Sprintf("-t nat -A POSTROUTING -s %s ! -o %s -j MASQUERADE",
subnet.String(), bridgeName)
    cmd := exec.Command("iptables", strings.Split(iptablesCmd, " ")...)
    // 执行 iptables 命令配置 SNAT 规则
    output, err := cmd.Output()
    if err != nil {
        log.Errorf("iptables Output, %v", output)
    }
    return nil
}
```

通过直接执行 iptables 命令，创建 SNAT 规则，只要是从这个网桥上出来的包，都会对其做源 IP 的转换，保证了容器经过宿主机访问到宿主机外部网络请求的包转换成机器的 IP，从而能正确的送达和接收。

6.4.3 Bridge Driver Delete 实现

6.4.2 小节介绍了 NetworkDriver 模块接口的定义。其中，删除网络的方法是 Delete，输入是网络对象，执行时会删除网络所对应的网络设备，而在 Bridge Driver 中，就是删除网络对应的 Linux Bridge 的设备，代码如下。

```
func (d *BridgeNetworkDriver) Delete(network Network) error {
    // 网络名即 Linux Bridge 的设备名
    bridgeName := network.Name
    // 通过 netlink 库的 LinkByName 找到网络对应的设备
    br, err := netlink.LinkByName(bridgeName)
    if err != nil {
        return err
    }
    // 删除网络对应的 Linux Bridge 设备
    return netlink.LinkDel(br)
}
```

6.4.4 测试

```
[~]$ # 调用 mydocker 创建网络
[~]$ sudo ./mydocker network create --driver bridge --subnet 192.168.10.1/24
testbridge
[~]$ # 调用 mydocker 列举创建的网络
[~]$ sudo ./mydocker network list
NAME         IpRange           Driver
testbridge   192.168.10.1/24   bridge
```

```
[~]$ # 查看创建出的 Bridge 设备
[~]$ ip link show dev testbridge
25: testbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN mode DEFAULT group default qlen 1000
    link/ether ba:fe:e5:bd:db:c8 brd ff:ff:ff:ff:ff:ff
[~]$ # 查看地址配置和路由配置
[~]$ ip addr show dev testbridge
25: testbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/ether ba:fe:e5:bd:db:c8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.1/24 scope global testbridge
       valid_lft forever preferred_lft forever
    inet6 fe80::b8fe:e5ff:febd:dbc8/64 scope link
       valid_lft forever preferred_lft forever
[~]$ ip route show dev testbridge
192.168.10.0/24  proto kernel  scope link  src 192.168.10.1

[~]$ # 查看 iptables 配置的 MASQUERADE 规则
[~]$ sudo iptables -t nat -vnL POSTROUTING
Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0
    0     0 MASQUERADE  all  --  *      !testbridge  192.168.10.0/24    0.0.0.0/0

[~]$ # 调用 mydocker 删除刚才创建的网络
[~]$ ./mydocker network remove testbridge
[~]$ # 列表中已经看不到网络
[~]$ ./mydocker network list
NAME        IpRange     Driver
[~]$ # 看到网络对应的设备也被删除
[~]$ ip addr show dev testbridge
Device "testbridge" does not exist.
```

6.5　在 Bridge 网络创建容器

本节代码获取：

```
git clone https://github.com/xianlubird/mydocker.git
git checkout code-6.5
```

在上一节中，使用 Go 配置 Linux Bridge 和 iptables 实现了容器网络的创建，而怎样配置容器的网络端点才能让容器通过这个网桥互相连接及与外部连接呢？在这一节中，会通过挂

载容器端点实现容器的互相通信及容器外部通信，最终实现通过 mydocker run -p 80:80 -net testbridge xxx 命令使容器加入网络及端口映射。

6.5.1 挂载容器端点的流程

在本章的第一节中，曾手动配置过容器的网络和容器的端点，先来回忆一下需要哪些步骤，如图 6.10 所示。

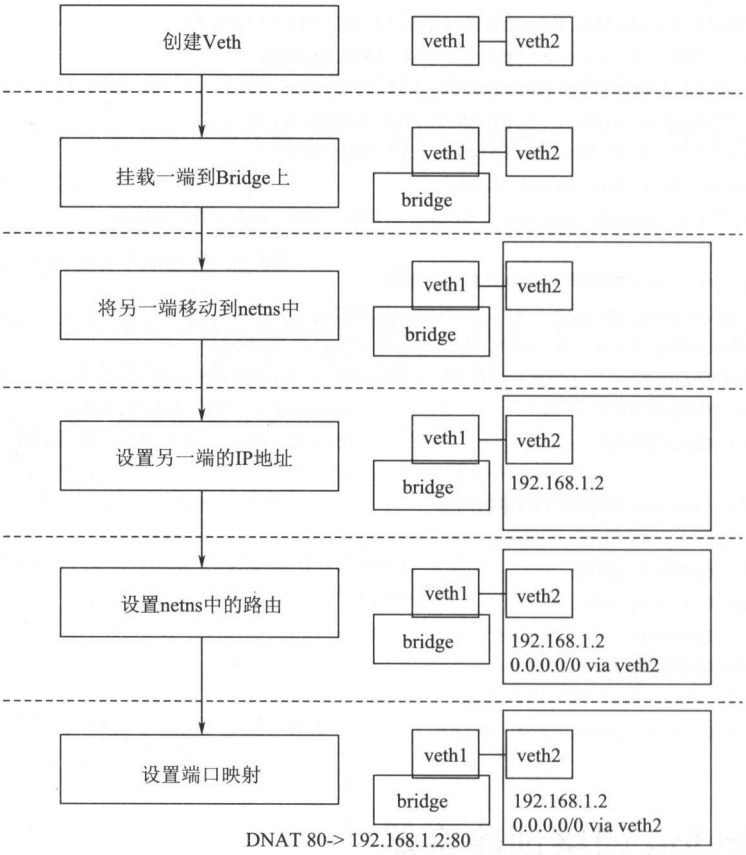


图 6.10

注意：图中的 netns 是 Net Namespace 的缩写。

从图 6.10 所示的流程中可以看到，在容器的 Net Namespace 中，就可以通过容器的 Veth 直接与挂载在同一个 Bridge 上的容器通信，以及通过 Bridge 上创建的 iptables 的 MASQUERADE 规则访问外部网络，同时，外部也可以通过机器上的端口经过 iptables 的 DNAT 的转发访问容器内部。

那么，下面就用代码来实现上面这些流程，给容器“插上网线”。

```
// 连接容器到之前创建的网络 mydocker run -net testnet -p 8080:80 xxxx
func Connect(networkName string, cinfo *container.ContainerInfo) error {
    // 从 networks 数组中取到网络的配置信息，如果找不到网络则返回错误
    network, ok := networks[networkName]
    if !ok {
        return fmt.Errorf("No Such Network: %s", networkName)
    }

    // 从网络的 IP 段中，分配容器 IP 地址
    ip, err := ipAllocator.Allocate(network.IpRange)
    if err != nil {
        return err
    }

    // 创建网络端点，设置网络端点的 IP、网络和端口映射信息，供下面的配置调用
    ep := &Endpoint{
        ID: fmt.Sprintf("%s-%s", cinfo.Id, networkName),
        IPAddress: ip,
        Network: network,
        PortMapping: cinfo.PortMapping,
    }
    // 调用网络对应的网络驱动挂载，配置网络端点
    if err = drivers[network.Driver].Connect(network, ep); err != nil {
        return err
    }
    // 到容器的 Namespace 中配置容器网络、设备 IP 地址和路由信息
    if err = configEndpointIpAddressAndRoute(ep, cinfo); err != nil {
        return err
    }

    // 配置端口映射信息，例如 mydocker run -p 8080:80
    return configPortMapping(ep, cinfo)
}
```

上面是对挂载容器端点流程的调用分解。其中，IP 地址分配在前 6.4.1 小节中已经介绍过了，下面以 Bridge 网络驱动为例介绍一下如何连接网络端点，如何对容器的 Net Namespace 做配置，以及如何映射容器端口到宿主机。

连接容器网络端点到 Linux Bridge

就像之前手动配置容器网络那样，Bridge 网络驱动会首先创建一个 Veth，并将一端挂载到

网络对应的 Linux Bridge 上，这样另外一端网络上的包就会通过 Linux Bridge 通信，便实现了容器网络端点的创建和挂载，代码如下。

```
// 连接一个网络和网络端点
func (d *BridgeNetworkDriver) Connect(network *Network, endpoint *Endpoint) error {
    // 获取网络名，即 Linux Bridge 的接口名
    bridgeName := network.Name
    // 通过接口名获取到 Linux Bridge 接口的对象和接口属性
    br, err := netlink.LinkByName(bridgeName)
    if err != nil {
        return err
    }

    // 创建 Veth 接口的配置
    la := netlink.NewLinkAttrs()
    // 由于 Linux 接口名的限制，名字取 endpoint ID 的前 5 位
    la.Name = endpoint.ID[:5]
    // 通过设置 Veth 接口的 master 属性，设置这个 Veth 的一端挂载到网络对应的 Linux Bridge 上
    la.MasterIndex = br.Attrs().Index

    // 创建 Veth 对象，通过 PeerName 配置 Veth 另外一端的接口名
    // 配置 Veth 另外一端的名字 cif-{endpoint ID 的前 5 位 }
    endpoint.Device = netlink.Veth{
        LinkAttrs: la,
        PeerName:  "cif-" + endpoint.ID[:5],
    }

    // 调用 netlink 的 LinkAdd 方法创建出这个 Veth 接口
    // 因为上面指定了 link 的 MasterIndex 是网络对应的 Linux Bridge
    // 所以 Veth 的一端就已经挂载到了网络对应的 Linux Bridge 上
    if err = netlink.LinkAdd(&endpoint.Device); err != nil {
        return fmt.Errorf("Error Add Endpoint Device: %v", err)
    }

    // 调用 netlink 的 LinkSetUp 方法，设置 Veth 启动
    // 相当于 ip link set xxx up 命令
    if err = netlink.LinkSetUp(&endpoint.Device); err != nil {
        return fmt.Errorf("Error set Endpoint Device up: %v", err)
    }
    return nil
}
```

通过调用 Bridge 驱动中的 Connect 方法，容器的网络端点已经挂载到了 Bridge 网络的 Linux Bridge 上。那么，下一步就是配置网络端点的另外一端，即容器的 Net Namespace 那一端。

配置容器 Namespace 中的网络设备及路由

容器有自己独立的 Net Namespace，需要将网络端点的 Veth 设备的另外一端移到这个 Net Namespace 中并配置，才能给这个容器“插上网线”，下面将配置网络端点的地址和路由。

```
// 配置容器网络端点的地址和路由
func configEndpointIpAddressAndRoute(ep *Endpoint, cinfo *container.ContainerInfo)
error {
    // 通过网络端点中“Veth”的另一端
    peerLink, err := netlink.LinkByName(ep.Device.PeerName)
    if err != nil {
        return fmt.Errorf("fail config endpoint: %v", err)
    }

    // 将容器的网络端点加入到容器的网络空间中
    // 并使这个函数下面的操作都在这个网络空间中进行
    // 执行完函数后，恢复为默认的网络空间，具体实现下面再做介绍
    defer enterContainerNetns(&peerLink, cinfo)()

    // 获取到容器的 IP 地址及网段，用于配置容器内部接口地址
    // 比如容器 IP 是 192.168.1.2，而网络的网段是 192.168.1.0/24
    // 那么这里产出的 IP 字符串就是 192.168.1.2/24，用于容器内 Veth 端点配置
    interfaceIP := *ep.Network.IpRange
    interfaceIP.IP = ep.IPAddress
    // 调用 setInterfaceIP 函数设置容器内 Veth 端点的 IP
    // 这个函数，在上一节配置 Bridge 时有介绍其实现
    if err = setInterfaceIP(ep.Device.PeerName, interfaceIP.String()); err != nil {
        return fmt.Errorf("%v,%s", ep.Network, err)
    }

    // 启动容器内的 Veth 端点
    if err = setInterfaceUP(ep.Device.PeerName); err != nil {
        return err
    }

    // Net Namespace 中默认本地地址 127.0.0.1 的“lo”网卡是关闭状态的
    // 启动它以保证容器访问自己的请求
    if err = setInterfaceUP("lo"); err != nil {
        return err
    }

    // 设置容器内的外部请求都通过容器内的 Veth 端点访问
    // 0.0.0.0/0 的网段，表示所有的 IP 地址段
    _, cidr, _ := net.ParseCIDR("0.0.0.0/0")
```

```
    // 构建要添加的路由数据，包括网络设备、网关 IP 及目的网段
    // 相当于 route add -net 0.0.0.0/0 gw {Bridge 网桥地址} dev {容器内的 Veth 端点设备}
    defaultRoute := &netlink.Route{
        LinkIndex: peerLink.Attrs().Index,
        Gw: ep.Network.IpRange.IP,
        Dst: cidr,
    }

    // 调用 netlink 的 RouteAdd，添加路由到容器的网络空间
    // RouteAdd 函数相当于 route add 命令
    if err = netlink.RouteAdd(defaultRoute); err != nil {
        return err
    }

    return nil
}
```

进入容器 Net Namespace

上面调用了 enterContainerNetns 的函数，使容器网络端点的 Veth 容器端，以及后续的配置都在容器的 Net Namespace 中执行。这是如何做到的呢？下面来看一下实现。

```
// 将容器的网络端点加入到容器的网络空间中
// 并锁定当前程序所执行的线程，使当前线程进入到容器的网络空间
// 返回值是一个函数指针，执行这个返回函数才会退出容器的网络空间，回归到宿主机的网络空间
// 这个函数中引用了之前介绍的 github.com/vishvananda/netns 类库来做 Namespace 操作
func enterContainerNetns(enLink *netlink.Link, cinfo *container.ContainerInfo)
func() {
    // 找到容器的 Net Namespace
    // /proc/[pid]/ns/net 打开这个文件的文件描述符就可以来操作 Net Namespace
    // 而 ContainerInfo 中的 PID，即容器在宿主机上映射的进程 ID
    // 它对应的 /proc/[pid]/ns/net 就是容器内部的 Net Namespace
    f, err := os.OpenFile(fmt.Sprintf("/proc/%s/ns/net", cinfo.Pid), os.O_RDONLY, 0)
    if err != nil {
        logrus.Errorf("error get container net namespace, %v", err)
    }

    // 取到文件的文件描述符
    nsFD := f.Fd()

    // 锁定当前程序所执行的线程，如果不锁定操作系统线程的话
    // Go 语言的 goroutine 可能会被调度到别的线程上去
    // 就不能保证一直在所需要的网络空间中了
    // 所以调用 runtime.LockOSThread 时要先锁定当前程序执行的线程
```

```
    runtime.LockOSThread()

    // 修改网络端点 Veth 的另外一端，将其移动到容器的 Net Namespace 中
    if err = netlink.LinkSetNsFd(*enLink, int(nsFD)); err != nil {
        logrus.Errorf("error set link netns , %v", err)
    }

    // 通过 netns.Get 方法获得当前网络的 Net Namespace
    // 以便后面从容器的 Net Namespace 中退出，回到原本网络的 Net Namespace 中
    orignS, err := netns.Get()
    if err != nil {
        logrus.Errorf("error get current netns, %v", err)
    }

    // 调用 netns.Set 方法，将当前进程加入容器的 Net Namespace
    if err = netns.Set(netns.NsHandle(nsFD)); err != nil {
        logrus.Errorf("error set netns, %v", err)
    }
    // 返回之前 Net Namespace 的函数
    // 在容器的网络空间中，执行完容器配置之后调用此函数就可以将程序恢复到原生的 Net Namespace
    return func () {
        // 恢复到上面获取到的之前的 Net Namespace
        netns.Set(orignS)
        // 关闭 Namespace 文件
        orignS.Close()
        // 取消对当前程序的线程锁定
        runtime.UnlockOSThread()
        // 关闭 Namespace 文件
        f.Close()
    }
}
```

以上即是如何使当前程序的执行环境进入到容器的 Net Namespace 用于配置容器地址和路由等的操作。当要进入容器的 Net Namespace 时，只需要调用 defer enterContainerNetns(&peerLink, cinfo)()，则在当前函数体结束之前都会在容器的 Net Namespace 中。在调用 enterContainerNetns(&peerLink, cinfo)() 时会使当前执行的函数进入容器的 Net Namespace，而用了 defer 关键字后会在函数体结束时执行返回的恢复函数指针，并会在函数结束后恢复到之前所在的网络空间。

配置宿主机到容器的端口映射

现在的容器已经有了自己的网络空间和地址，但是这个地址是访问不到宿主机外部的，所

以需要将容器的地址映射到宿主机上。可以通过 iptables 的 DNAT 规则来实现宿主机上的请求转发到容器上。

```
// 配置端口映射
func configPortMapping(ep *Endpoint, cinfo *container.ContainerInfo) error {
    // 遍历容器端口映射列表
    for _, pm := range ep.PortMapping {
        // 分割成宿主机的端口和容器的端口
        portMapping :=strings.Split(pm, ":")
        if len(portMapping) != 2 {
            logrus.Errorf("port mapping format error, %v", pm)
            continue
        }

        // 由于 iptables 没有 Go 语言版本的实现，所以采用 exec.Command 的方式直接调用命令配置
        // 在 iptables 的 PREROUTING 中添加 DNAT 规则
        // 将宿主机的端口请求转发到容器的地址和端口上
        iptablesCmd := fmt.Sprintf("-t nat -A PREROUTING -p tcp -m tcp --dport %s
-j DNAT --to-destination %s:%s",
            portMapping[0], ep.IPAddress.String(), portMapping[1])
        // 执行 iptables 命令，添加端口映射转发规则
        cmd := exec.Command("iptables", strings.Split(iptablesCmd, " ")...)
        output, err := cmd.Output()
        if err != nil {
            logrus.Errorf("iptables Output, %v", output)
            continue
        }
    }
    return nil
}
```

6.5.2 测试

6.5.1 小节中完成了对容器端点的配置流程，就算是真正给容器“插上了网线”。下面来测试一下容器的连接。首先，创建一个供容器连接的网络，用于让容器挂载。

```
[~]$ mydocker network create --driver bridge --subnet 192.168.10.1/24 testbridge
```

容器与容器互联

分别在创建网络上启动两个容器，并拿到第一个容器的 IP，在第二个容器中去访问。

```
[~]$ mydocker run -ti -net testbridge busybox sh
{"level":"info","msg":"createTty true","time":"2017-01-10T00:51:20+08:00"}
{"level":"info","msg":"init come on","time":"2017-01-10T00:51:20+08:00"}
```

```
{"level":"info","msg":"command all is sh","time":"2017-01-10T00:51:20+08:00"}
{"level":"info","msg":"Current location is /root/mnt/3286310293","time":"2017-01-
10T00:51:20+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-10T00:51:20+08:00"}
/ # ifconfig
cif-32863 Link encap:Ethernet  HWaddr C6:76:00:7B:27:D8
          inet addr:192.168.10.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::c476:ff:fe7b:27d8/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:20 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:775 (775.0 B)  TX bytes:1324 (1.2 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:828 (828.0 B)  TX bytes:828 (828.0 B)
```

这个容器的 IP 地址是 192.168.10.2，下面尝试一下在另外一个容器中连接这个容器。

```
[~]$ mydocker run -ti -net testbridge busybox sh
{"level":"info","msg":"createTty true","time":"2017-01-10T01:00:04+08:00"}
{"level":"info","msg":"init come on","time":"2017-01-10T01:00:04+08:00"}
{"level":"info","msg":"command all is sh","time":"2017-01-10T01:00:04+08:00"}
{"level":"info","msg":"Current location is /root/mnt/8202025412","time":"2017-01-
10T01:00:04+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-10T01:00:04+08:00"}
/ # # ifconfig 查看这个容器的 IP 地址是 192.168.10.3
/ # ifconfig
cif-06865 Link encap:Ethernet  HWaddr B6:FE:67:F1:21:AE
          inet addr:192.168.10.3  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::b4fe:67ff:fef1:21ae/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:70 (70.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
```

```
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
/ # # 访问上面创建的那个容器的 IP 地址
/ # ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2): 56 data bytes
64 bytes from 192.168.10.2: seq=0 ttl=64 time=0.111 ms
64 bytes from 192.168.10.2: seq=1 ttl=64 time=0.093 ms
64 bytes from 192.168.10.2: seq=2 ttl=64 time=0.094 ms
^C
--- 192.168.10.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.093/0.099/0.111 ms
```

由以上结果可以看到，两个容器可以通过这个网络互相连通。

容器访问外部网络

```
[~]$ mydocker run -ti -net testbridge busybox sh
{"level":"info","msg":"createTty true","time":"2017-01-10T01:00:04+08:00"}
{"level":"info","msg":"init come on","time":"2017-01-10T01:00:04+08:00"}
{"level":"info","msg":"command all is sh","time":"2017-01-10T01:00:04+08:00"}
{"level":"info","msg":"Current location is /root/mnt/8202025412","time":"2017-01-
10T01:00:04+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-10T01:00:04+08:00"}
/ # # 由于容器中默认没有配置 DNS 服务器，所以先来配置一下 DNS 服务器
/ # echo "nameserver 10.143.22.118" > /etc/resolv.conf
/ # ping baidu.com
PING baidu.com (180.149.132.47): 56 data bytes
64 bytes from 180.149.132.47: seq=0 ttl=52 time=23.725 ms
64 bytes from 180.149.132.47: seq=1 ttl=52 time=24.948 ms
64 bytes from 180.149.132.47: seq=2 ttl=52 time=23.780 ms
^C
--- baidu.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 23.725/24.151/24.948 ms
```

容器也可以通过 Bridge 网络的网桥访问外部网络。

容器映射端口到宿主机上供外部访问

通过 mydocker run -p 80:80 的方式将容器中的 80 端口映射到宿主机上。

```
[~]$ mydocker run -ti -p 80:80 -net testbridge busybox sh
{"level":"info","msg":"createTty true","time":"2017-01-10T00:51:20+08:00"}
{"level":"info","msg":"init come on","time":"2017-01-10T00:51:20+08:00"}
{"level":"info","msg":"command all is sh","time":"2017-01-10T00:51:20+08:00"}
{"level":"info","msg":"Current location is /root/mnt/3286310293","time":"2017-01-
10T00:51:20+08:00"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-10T00:51:20+08:00"}
/ # nc -lp 80
```

然后访问宿主机的 80 端口，看是否能转发到容器中。

```
[~]$ telnet 118.178.142.223 80
Trying 118.178.142.223...
Connected to 118.178.142.223.
Escape character is '^]'.
hello world
^]
telnet> q
Connection closed.
```

用 telnet 对宿主机的端口发送一个 hello world，容器中立即用 nc 监听就收到了这段问候。

```
/ # nc -lp 80
hello world
/ #
```

6.6　容器跨主机网络

在 6.5 节中，实现了单个宿主机上容器网络的互联互通，以及容器与外部网络的互通。但是，由于 Linux Bridge 没有办法路由到宿主机外部，所以，不同宿主机 Bridge 上的容器还是得通过映射到端口的方式来实现互相访问。那么，多个容器就没办法同时使用同一个端口了，而且访问的地址也不是容器自身的 IP 地址，并且暴漏到宿主机端口上，可以直接提供宿主机的地址来访问容器的服务，也不安全。

那么，要怎么实现让容器能够通过各自的地址直接通信的功能呢？这一节主要介绍一下如何做到容器跨主机网络的方案，跨主机网络涉及的状态处理复杂，所以这里就不涉及具体实现了。

6.6.1 跨主机容器网络的 IPAM

在 6.2 节中，介绍了 IPAM，通过将 IP 地址分配信息的位图存放在文件中，实现了容器和网关的 IP 地址分配。但通常情况下，没办法在多个宿主机上使用同一个文件来做 IP 地址分配，如果每个机器只负责容器网络在自己宿主机上的 IP 地址分配，那么就可能会造成不同机器上分配到的容器 IP 地址重复的问题。如果同一个容器网络中的 IP 重复了，就会导致不可预期的访问问题。

所以，通常会采用中心化的一致性 KV-store 来作为记录 IP 地址分配的存储。对于跨主机的容器网络，把 IP 地址分配信息的位图存放在中心的一致性 KV-store 中，来保证每个宿主机上分配给容器的 IP 不冲突。这里可能就有疑问了。什么是一致性 KV-store？如何使用一致性 KV-store 来存储和管理多个节点上共享的信息？

一致性 KV-store

一致性 KV-store 的 KV 是指 key-value，相当于写代码时常用到的 map 类型中的 key-value，而 KV-store 就是存储这种 key-value 对应关系的一种数据库。

比如，一个容器网络的网段为 192.168.0.0/30，在 6.4 节中介绍 IPAM 的时候，存储的这个地址分配信息是“192.168.0.0/30”，“0000”用来代表这个网段的 4 个 IP 都未被分配，“192.168.0.0/30”就可以作为网段的 key，而 value 是分配信息的位图“0000”。前面通过 `json` 对象的方式将 key-value 存到了文件中，现在为了让多个宿主机都可以同时访问到 IP 的分配信息，就可以存储到中心化的一致性 KV-store 中，保证每个宿主机对分配信息的访问。

而当多个宿主机同时访问时就会引起并发的问题，比如在两个宿主机上同时启动容器，就会同时分配 IP 地址，这样就可能会分配到同样的 IP 地址，那么一致性 KV-store 的一致性就派上用场了。通过一致性 KV-store，可以保证每个宿主机上 KV 信息的强一致性，不会出现并发时分配到同样 IP 地址的问题。通常会用如下两种方式实现节点一致。

- 全局锁，很多一致性 KV-store 的实现支持对某一个 key 或者路径锁的方法，拿到这个锁的客户端才能修改对应的数据，比如在分配 IP 之前先拿存放分配信息对应的 `key` 锁，分配完成并写入后再释放锁。这样就能保证同一时间只有一个宿主机在分配地址，确保地址不重复。
- Compare And Swap，有的一致性 KV-store 会实现 Compare And Swap（CAS）的原子方法。通过 CAS 方法，每个宿主机在写入 IP 分配的信息时，会判断分配过程中这个 IP 分配信息的 key 是不是被修改过。如果被修改过，则重新获取信息并重新分配 IP 后再重试。这种

方式能够保证在同一时间只有一个宿主机上分配地址成功，同样可以保证地址不重复。

常见的一致性 KV-store 有 etcd、consul、zookeeper 等，都可以用来实现跨主机网络的 IP 地址分配需求。

6.6.2　跨主机容器网络通信的常见实现方式

跨主机容器网络的地址分配问题解决了，前面的章节介绍了容器在宿主机上的 IP 是通过配置 route 的方式配置路由访问到的，但是在宿主机之间是访问不到这个地址的，那么跨主机容器之间的通信又要怎样实现呢？或者说怎么把容器的访问包通过宿主机的网络送达呢？通常会有以下两种实现方式。

- 封包。
- 路由。

封包

既然在宿主机之间不知道容器的地址要怎么路由和访问，就可以把容器之间的请求外面包装上宿主机的地址发送，这样跨主机的容器之间的通信就转换成了宿主机之间的通信，到达另外一个容器所在的宿主机后再解开外面的包装，拿到真正的容器请求，就能实现跨主机的容器间通信了。

如图 6.11 所示，将 172.18.0.2 的容器到 172.18.0.3 的容器请求封装成宿主机 192.168.0.2 到 192.168.0.3 的通信，然后将 192.168.0.3 的宿主机解开包后，就可以看到真正的目的地址 172.18.0.3，便实现了跨宿主机容器网络通信。

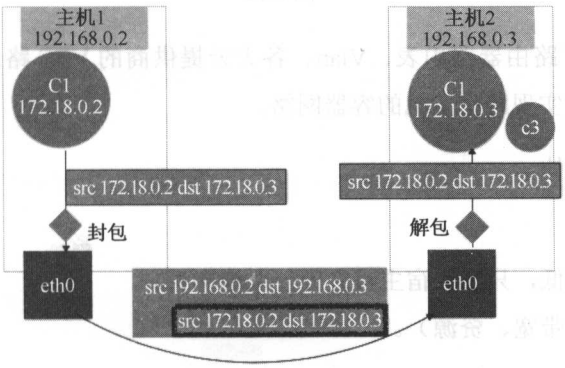


图 6.11

常见的封包技术有 Vxlan、Ipip-tunnel、GRE，或者自己的封包格式等，通过这些技术就可

以将容器间的请求封装成宿主机间的请求。

路由

还有一种实现跨主机容器网络通信的方式是路由，这种方式的原理是让宿主机的网络“知道”容器的地址要怎么路由、路由到哪台机器上，这种方式一般需要网络设备的支持，比如修改路由器的路由表，将容器 IP 地址的下一跳修改到这个容器所在的宿主机上，来送达跨主机容器间的请求。

如图 6.12 所示，容器 192.168.1.2 对容器 192.168.2.2 的访问请求到达宿主机的路由器时，路由器通过路由表找到这个 IP 的下一跳是主机 2，那么就把请求转发到主机 2 上，最终把跨主机容器间的请求送达，实现跨主机的容器间通信。

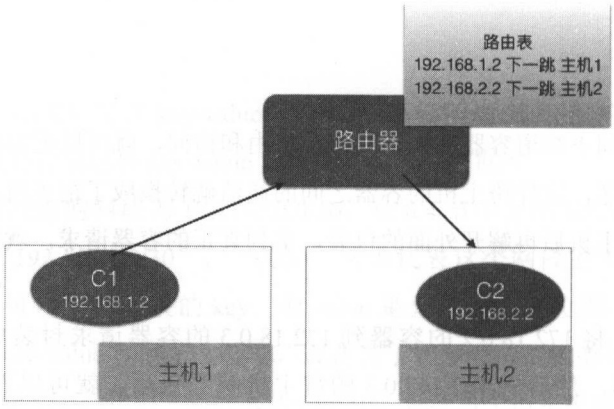


图 6.12

常见的路由技术有路由器路由表、Vlan、各大云提供商的 VPC 路由表等，通过对这些路由方式的配置，也可以实现跨宿主机的容器网络。

两种实现方式的对比

- 封包方式。
  - 基础设施要求低，只需要宿主机之间能联通即可。
  - 性能损耗大（带宽、资源）。
- 路由方式。
  - 无封包，性能好。
  - 对基础网络设施有要求 需要支持一些路由的配置或者特定的网络协议。

6.7　小结

在这一章中，首先手动给一个容器配置了网路，并通过这个过程了解了 Linux 虚拟网络设备和操作。然后构建了容器网络的概念模型和模块调用关系、IP 地址分配方案，以及网络模块的接口设计和实现，并且通过实现 Bridge 驱动给容器连上了“网线”。6.6 节中介绍了实现跨主机容器网络的手段和关键技术，通过这些技术可以实现跨主机容器网络的地址分配和互联互通。

第7章

高级实践

7.1 使用 mydocker 创建一个可访问的 nginx 容器

经过前面章节的开发，我们的容器已经具备了网络访问功能，容器新功能开发部分到此结束。但是经过如此漫长的开发，这个容器到底怎么样呢，能不能真正部署一个可以访问的程序呢？带着这个疑问开始本章的探索吧！本章首先会创建一个 nginx 容器，对外暴露一个可以访问的端口，请求会经过端口转发进入到 nginx 内部。是不是很期待？下面就开始吧。

7.1.1 获取 nginx tar 包

在开始运行前，首先需要获取 nginx 的文件系统。一种方式是基于最初的 busybox 自己安装 nginx，但是为了测试 mydocker 程序的通用性，使用 Docker 的 nginx 镜像来运行下面的例子。

```
## 运行一个 nginx 容器
root@vagrant-ubuntu-trusty-64:~# docker run -d nginx
e9b60bce992be132cb58d50c921c0d607705513d1853113a7e5b38783cae1a1d

## 查看一下运行容器
root@vagrant-ubuntu-trusty-64:~# docker ps
CONTAINER ID          IMAGE                  COMMAND                  CREATED
STATUS              PORTS               NAMES
e9b60bce992b         nginx                 "nginx -g 'daemon off"   3 seconds ago
Up 2 seconds        80/tcp, 443/tcp     infallible_sinoussi

## 使用 export 将文件系统打包成 tar 存储到本地
```

```
root@vagrant-ubuntu-trusty-64:~# docker export -o nginx.tar e9b60bce992b
## 可以看到 nginx.tar 已经存储到本地磁盘
root@vagrant-ubuntu-trusty-64:~# ll
total 183624
drwx------  5 root root      4096 Jan 10 11:07 ./
drwxr-xr-x 24 root root      4096 Jan 10 06:05 ../
-rw-------  1 root root     30072 Jan 10 08:23 .bash_history
-rw-r--r--  1 root root      3160 Dec 18 08:12 .bashrc
drwxr-xr-x  2 root root      4096 Jan 10 09:07 mnt/
-rw-------  1 root root        16 Jan 10 08:22 .mysql_history
-rw-------  1 root root 187953152 Jan 10 11:07 nginx.tar
-rw-r--r--  1 root root       140 Feb 20  2014 .profile
drwx------  2 root root      4096 Dec 18 06:10 .ssh/
-rw-------  1 root root      6403 Jan  9 14:48 .viminfo
drwxr-xr-x  2 root root      4096 Jan 10 09:07 writeLayer/
root@vagrant-ubuntu-trusty-64:~#
```

这样就获得了 nginx.tar。下面，容器就会基于这个镜像来运行。

7.1.2　构建自己的 nginx 镜像

Docker 默认的 nginx 镜像的一些配置是通用配置，但是我们期望去修改一下 nginx 的配置文件，顺便测试一下构建镜像功能。

```
## 运行 nginx 容器，为了方便 exec 进入容器，使用 top -b 命令常驻前台
./mydocker run -d --name mynginx nginx top -b
{"level":"info","msg":"createTty false","time":"2017-01-10T11:12:56Z"}
{"level":"info","msg":"command all is top -b","time":"2017-01-10T11:12:58Z"}

## 可以看到，容器已经处于运行状态
root@ubuntu:[mydocker]# ./mydocker ps
ID           NAME         PID         STATUS      COMMAND     CREATED
1916745081   mynginx      10786       running     top-b       2017-01-10 11:12:58
```

下面会执行 exec 进入到这个容器中，去修改一下默认的 nginx 配置文件。

```
## 进入容器内部
./mydocker exec mynginx sh
{"level":"info","msg":"container pid 10786","time":"2017-01-10T11:15:11Z"}
{"level":"info","msg":"command sh","time":"2017-01-10T11:15:11Z"}

## 查看一下当前容器内进程，可以看到运行的 top 命令
# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 11:12 ?        00:00:00 top -b
```

```
root         4     0  0 11:15 ?        00:00:00 sh -c sh
root         5     4  0 11:15 ?        00:00:00 sh
root         6     5  0 11:15 ?        00:00:00 ps -ef

# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var

## 进入到存储 nginx 配置文件的目录
# cd etc/nginx

# ls
conf.d  fastcgi_params  koi-utf  koi-win  mime.types  modules  nginx.conf  scgi_
params  uwsgi_params  win-utf

## 查看一下默认的 nginx 配置文件
# cat nginx.conf

user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;
```

```
    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
}

## 由于 nginx 默认是在后台运行，所以为了保证 nginx 可以在前台运行，需要添加一个参数
# echo "daemon off;" >> nginx.conf
```

如此 nginx 配置文件就已经修改完毕。下面 commit 一下这个容器，构建出来我们自己的镜像。

```
##commit 容器为我们自己的镜像
./mydocker commit mynginx[ 容器名 ] mynginx[ 期望构建的镜像名 ]

## 查看一下，发现 mynginx.tar 包已经构建成功
root@ubuntu:[mydocker]# ls /root/
mnt  mynginx.tar  nginx  nginx.tar  writeLayer
```

7.1.3　运行 mynginx 容器

下面就基于前面创建的镜像开始实验。因为需要暴露访问端口给外部，所以在真正运行容器之前，先创建一个网络来供这个容器使用。

```
## 创建网络
./mydocker network create --driver bridge --subnet 192.168.10.1/24 mynginxbridge

## 查看创建的网桥
root@ubuntu:[mydocker] ifconfig | grep mynginxbridge

mynginxbridge Link encap:Ethernet  HWaddr 4e:c3:6c:17:af:c4
      inet addr:192.168.10.11  Bcast:0.0.0.0  Mask:255.255.255.0
      inet6 addr: fe80::4cc3:6cff:fe17:afc4/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
```

下面开始创建容器。

```
## 创建容器，使用刚刚创建的 mynginxbridge 网桥，对外暴露 8888 端口作为访问端口
## 容器执行命令为 nginx
./mydocker run -d --name bird -net mynginxbridge -p 8888:80 mynginx nginx
```

```
{"level":"info","msg":"createTty false","time":"2017-01-10T11:30:04Z"}
{"level":"info","msg":"command all is nginx","time":"2017-01-10T11:30:07Z"}

## 查看一下宿主机进程，可以发现 nginx 进程已经开始运行
root@ubuntu:[mydocker]# ps -ef | grep nginx
root     10741 10727  0 11:07 ?        00:00:00 nginx: master process nginx -g
daemon off;
sshd     10761 10741  0 11:07 ?        00:00:00 nginx: worker process
root     10984     1  0 11:30 stderr   00:00:00 nginx: master process nginx
sshd     11017 10984  0 11:30 stderr   00:00:00 nginx: worker process
root     11019 10106  0 11:30 pts/2    00:00:00 grep --color=auto nginx

## 查看当前容器运行状态
root@ubuntu:[mydocker]# ./mydocker ps
ID           NAME         PID          STATUS       COMMAND      CREATED
4058959272   bird         10984        running      nginx        2017-01-10 11:30:06
```

可以发现，这时候容器已经正常运行。下面，exec 到容器内部查看一下容器被分配的 IP。

```
## 进入容器内部
./mydocker exec bird sh
{"level":"info","msg":"container pid 10984","time":"2017-01-10T11:34:28Z"}
{"level":"info","msg":"command sh","time":"2017-01-10T11:34:28Z"}

## 查看容器 IP，可以发现，容器 IP 为 192.168.10.12
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
39: cif-40589: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 76:cc:93:ef:66:11 brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.12/24 scope global cif-40589
       valid_lft forever preferred_lft forever
    inet6 fe80::74cc:93ff:feef:6611/64 scope link
       valid_lft forever preferred_lft forever
```

实验运行 nginx 容器的宿主机 IP 地址是 192.168.33.10，下面就在另外一台非宿主机的机器上，使用浏览器访问一下宿主机的 ip:port，查看容器的 nginx 是否已经成功映射到了宿主机

的 8888 端口对外访问。

如图 7.1 所示，成功地在外部访问到了宿主机的 8888 端口，这个请求被转发到了容器内部，由 nginx 容器提供了服务。至此，就成功地基于 Docker 的 nginx 镜像构建了我们自己的镜像，并且可以对外提供服务。

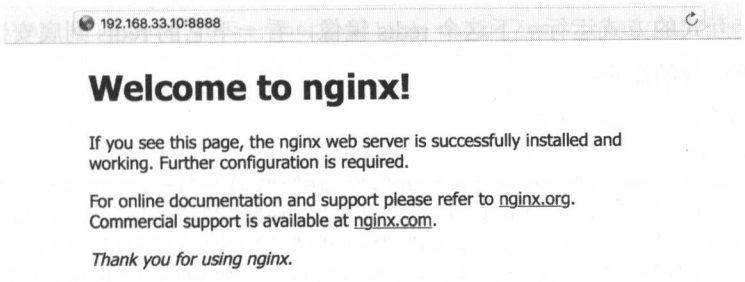


图 7.1

7.2　使用 mydocker 创建一个 flask + redis 的计数器

在 7.1 节中，创建了一个外部可访问的 nginx，但是那仅仅是一个容器，并没有提供两个容器之间互相访问的例子。本节就会创建这样一个例子，redis 和 flask 分别在不同的容器中，它们通过 IP 互相访问，flask 会把计数结果存储到 redis 中，并且进行累加，一次校验多容器互相连接访问的能力。

7.2.1　创建 redis 容器

首先，需要 redis 镜像，还是和上一章一样，直接使用 Docker 默认的 redis 镜像。

```
## 创建 redis 容器
docker run -d redis
7dcec56ac429e7563339e35912c2763eca2150dc057a3ef9d688a1c91ab342ee

## 查看当前容器
root@ubuntu:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS
PORTS               NAMES
7dcec56ac429   redis          "docker-entrypoint.sh"    3 seconds ago        Up 2
seconds         6379/tcp            high_stallman
```

```
## 将 redis 容器打包成 tar
root@ubuntu:~# docker export -o redis.tar 7dcec56ac429

## 可以看到，/root 下已经有了 redis.tar 包
root@ubuntu:~# ls /root/
mnt  redis.tar  writeLayer
```

首先，以交互式的方式运行一下这个 redis 镜像，看一下它的 redis 到底安装在什么地方，便于确定初始化运行的命令。

```
## 以交互式的方式运行 redis 容器
./mydocker run -ti redis sh
{"level":"info","msg":"createTty true","time":"2017-01-10T12:11:16Z"}
{"level":"info","msg":"command all is sh","time":"2017-01-10T12:11:17Z"}
{"level":"info","msg":"init come on","time":"2017-01-10T12:11:17Z"}
{"level":"info","msg":"Current location is /root/mnt/0106800024","time":"2017-01-
10T12:11:17Z"}
{"level":"info","msg":"Find path /bin/sh","time":"2017-01-10T12:11:17Z"}

## 查找 redis 的安装位置
# find / -name redis*
/usr/local/bin/redis-sentinel
/usr/local/bin/redis-check-rdb
/usr/local/bin/redis-benchmark
/usr/local/bin/redis-server
/usr/local/bin/redis-cli
/usr/local/bin/redis-check-aof
#
```

这里可以看到，redis-server 就是想要找到的命令，下面就以这个命令来运行 redis 容器。在此之前，由于 redis 也需要 IP 来对外提供服务，因此需要先创建一个网络。

```
## 创建名为 myflask 的网桥，子网段为 192.168.20.1/24
./mydocker network create --driver bridge --subnet 192.168.20.1/24 myflask

## 查看一下网桥的创建情况
root@ubuntu:[mydocker]# ./mydocker network list
NAME        IpRange           Driver
myflask     192.168.20.1/24   bridge

root@ubuntu:[mydocker]# ifconfig | grep myflask
myflask   Link encap:Ethernet  HWaddr ae:84:ac:1b:45:29
      inet addr:192.168.20.1  Bcast:0.0.0.0  Mask:255.255.255.0
      inet6 addr: fe80::ac84:acff:fe1b:4529/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
        TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B)  TX bytes:508 (508.0 B)
```

可以看到，网桥已经创建成功。下面就来启动 redis 容器。

```
## 创建 redis 容器，加入 myflask 网络，运行命令为 /usr/local/bin/redis-server
./mydocker run -d --name redis -net myflask redis /usr/local/bin/redis-server
{"level":"info","msg":"createTty false","time":"2017-01-10T12:16:19Z"}
{"level":"info","msg":"command all is /usr/local/bin/redis-server","time":"2017-
01-10T12:16:19Z"}

## 查看 redis 进程
root@ubuntu:[mydocker]# ps -ef | grep redis
root     11932     1  0 12:16 pts/2    00:00:00 /usr/local/bin/redis-server *:6379
root     11960 10106  0 12:16 pts/2    00:00:00 grep --color=auto redis

## 查看容器运行状态
root@ubuntu:[mydocker]# ./mydocker ps
ID                  NAME             PID              STATUS          COMMAND
CREATED
6564910843    redis         11932         running       /usr/local/bin/redis-server
2017-01-10 12:16:19

## 查看容器运行日志，可以发现，redis 已经启动成功
root@ubuntu:[mydocker]# ./mydocker logs redis
{"level":"info","msg":"init come on","time":"2017-01-10T12:16:19Z"}
{"level":"info","msg":"Current location is /root/mnt/redis","time":"2017-01-
10T12:16:19Z"}
{"level":"info","msg":"Find path /usr/local/bin/redis-server","time":"2017-01-
10T12:16:19Z"}
1:C 10 Jan 12:16:19.326 # Warning: no config file specified, using the default
config. In order to specify a config file use /usr/local/bin/redis-server /path/
to/redis.conf
1:M 10 Jan 12:16:19.327 * Increased maximum number of open files to 10032 (it was
originally set to 1024).
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 3.2.6 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 1
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
```

```
|    `-._`-._        _.-'_.-'    |          http://redis.io
 `-._    `-._`-.__.-'_.-'    _.-'
|`-._`-._    `-.__.-'    _.-'_.-'|
|    `-._`-._        _.-'_.-'    |
 `-._    `-._`-.__.-'_.-'    _.-'
     `-._    `-.__.-'    _.-'
         `-._        _.-'
             `-.__.-'
```

```
1:M 10 Jan 12:16:19.328 # WARNING: The TCP backlog setting of 511 cannot be
enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
1:M 10 Jan 12:16:19.328 # Server started, Redis version 3.2.6
1:M 10 Jan 12:16:19.328 # WARNING overcommit_memory is set to 0! Background
save may fail under low memory condition. To fix this issue add 'vm.overcommit_
memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl
vm.overcommit_memory=1' for this to take effect.
1:M 10 Jan 12:16:19.328 * The server is now ready to accept connections on port
6379
```

下面进入 redis 容器中，来获取 redis 被分配的 IP。

```
## 进入 redis 容器
./mydocker exec redis sh
{"level":"info","msg":"container pid 11932","time":"2017-01-10T12:20:56Z"}
{"level":"info","msg":"command sh","time":"2017-01-10T12:20:57Z"}
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
48: cif-65649: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether fa:c7:6c:97:66:94 brd ff:ff:ff:ff:ff:ff
    inet 192.168.20.2/24 scope global cif-65649
       valid_lft forever preferred_lft forever
    inet6 fe80::f8c7:6cff:fe97:6694/64 scope link
       valid_lft forever preferred_lft forever
```

可以发现，redis 的 IP 为 192.168.20.2。

7.2.2　制作 flask 镜像

下面基于 Docker 的 Python 2.7 镜像来构建 flask 镜像。

```
## 运行 Python 容器
docker run -d python:2.7 top -b
Unable to find image 'python:2.7' locally
2.7: Pulling from library/python
75a822cd7888: Already exists
57de64c72267: Pull complete
4306be1e8943: Pull complete
871436ab7225: Pull complete
37c937b0ca47: Pull complete
608a51124afe: Pull complete
086c59e7b25f: Pull complete
Digest: sha256:b21b2ba9b8bb8c8acc52915ac9c35be0bc08a9a7cb0a7852f8d2a0c5d4887f72
Status: Downloaded newer image for python:2.7
8761536753e8c7734237e71d272e8f11754901dd3525f176ba2d687b88ce8c78


## 查看容器运行状态
root@ubuntu:[mydocker]# docker ps
CONTAINER ID          IMAGE                    COMMAND                  CREATED
STATUS              PORTS               NAMES
8761536753e8          python:2.7             "top -b"              3 seconds ago
Up 2 seconds                              cocky_hamilton

## 打包 Python 镜像
root@ubuntu:~# docker export -o python.tar 8761536753e8[ 容器 ID]

## 可以看到，python.tar 已经被存储到磁盘上
root@ubuntu:~# ls /root/
mnt  python.tar  redis  redis.tar  writeLayer
```

接下来先以交互式命令进入myflask容器中，安装需要的依赖。因为容器内需要安装依赖等，所以也需要将容器加入到网络中。

```
## 用交互式命令创建 myflask 容器
./mydocker run -ti -net myflask python sh
{"level":"info","msg":"createTty true","time":"2017-01-10T14:55:00Z"}
{"level":"info","msg":"command all is bash","time":"2017-01-10T14:55:00Z"}
{"level":"info","msg":"init come on","time":"2017-01-10T14:55:00Z"}
{"level":"info","msg":"Current location is /root/mnt/7428112071","time":"2017-01-
10T14:55:00Z"}
{"level":"info","msg":"Find path /bin/bash","time":"2017-01-10T14:55:00Z"}
```

```
## 设置容器的 DNS 解析地址
# echo "nameserver 114.114.114.114" >> /etc/resolv.conf

## 查看系统版本
# cat /etc/issue
Debian GNU/Linux 8 \n \l
```

由于需要在容器中安装 Vim 来进行一些文本操作，因此使用阿里云的 Debian 镜像加速地址。

```
## 设置阿里云的加速地址源
# echo "deb http://mirrors.aliyun.com/debian/ jessie main non-free contrib" > /
etc/apt/sources.list

## 安装 Vim
# apt-get update && apt-get install -y vim
```

因为需要使用 pip 安装 flask 和其他依赖，所以为了加快安装速度，需要配置一下 pip 的加速源。

```
## 创建 pip 的配置文件
# mkdir ~/.pip
# cd ~/.pip/
# touch pip.conf
# vim pip.conf

## 将配置文件的内容写入，所有内容如下
[global]
index-url = http://mirrors.aliyun.com/pypi/simple/

[install]
trusted-host=mirrors.aliyun.com
```

下面就可以使用 pip 安装依赖了。首先要安装 flask。

```
# pip install flask
Successfully built itsdangerous MarkupSafe
Installing collected packages: itsdangerous, MarkupSafe, Jinja2, Werkzeug, click,
flask
Successfully installed Jinja2-2.9.3 MarkupSafe-0.23 Werkzeug-0.11.15 click-6.7
flask-0.12 itsdangerous-0.24
```

再安装 redis 依赖包。

```
# pip install redis
Collecting redis
```

```
Installing collected packages: redis
Successfully installed redis-2.10.5
```

所有的依赖安装完成之后，来看一下代码。在 /root/ 下创建 app.py，内容如下。

```
import os
from flask import Flask
from redis import Redis

os.system('mknod -m 644 /dev/urandom c 1 9')
app = Flask(__name__)
redis = Redis(host='192.168.20.2', port=6379)


@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello World! I have been seen %s times.' % redis.get('hits')


if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

代码很简单，就是连接前面创建好的、IP 为 192.168.20.2 的 redis 容器。然后启动一个 web server，浏览器每访问一次，就会在 redis 中记录一次，依次累加。

由于 flask 在启动的时候需要 /dev/urandom 文件，因此在程序启动时，执行下面这行代码，来创建 /dev/urandom 这个文件。

```
os.system('mknod -m 644 /dev/urandom c 1 9')
```

至此，镜像就算制作完成了。下面另外开启一个窗口，将这个正在运行中的容器打包。

```
# ./mydocker ps
ID          NAME        PID     STATUS    COMMAND                      CREATED
7428112071  7428112071  13476   running   bash                         2017-01-10
14:55:00
6564910843  redis       11932   running   /usr/local/bin/redis-server  2017-01-10
12:16:19
```

NAME 为 7428112071，就是目前正在运行的 myflask 容器，下面对其执行 commit 操作。

```
./mydocker commit 7428112071 myflask
## 可以看到，myflask.tar 已经生成了
root@ubuntu:[mydocker]# ls /root/
mnt  myflask.tar  python  python.tar  redis  redis.tar  writeLayer
```

7.2.3 创建 myflask 容器

下面基于前面创建的 myflask 镜像来构建容器。

```
## 创建名为 flask 的容器，使用 myflask 网桥，将容器内的 5000 端口暴露到主机的 5000 端口
## 并且执行初始化命令
./mydocker run -d --name flask -net myflask -p 5000:5000 myflask python /root/
app.py
{"level":"info","msg":"createTty false","time":"2017-01-10T15:59:55Z"}
{"level":"info","msg":"command all is python /root/app.py","time":"2017-01-
10T15:59:55Z"}

## 可以看到 Python 进程已经开始运行了
root@ubuntu:[mydocker]# ps -ef | grep python
root     15283     1  2 15:59 pts/2    00:00:00 python /root/app.py
root     15310 15283  2 15:59 pts/2    00:00:00 /usr/local/bin/python/root/app.py
root     15316 10106  0 15:59 pts/2    00:00:00 grep --color=auto python

## 查看容器状态
root@ubuntu:[mydocker]# ./mydocker ps
ID          NAME   PID    STATUS   COMMAND                    CREATED
3302419117  flask    15283   running   python/root/app.py              2017-01-10
15:59:55
6564910843 redis  11932  running  /usr/local/bin/redis-server 2017-01-10 12:16:19
```

在外部访问宿主机的 5000 端口，宿主机 IP 为 192.168.33.10。

从图 7.2 到图 7.4 中可以看到，宿主机能够被正常访问，而且每刷新一次，计数器都会加 1。这样就演示了两个容器之间互相访问和配合服务的过程。

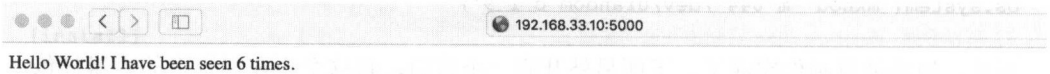


图 7.2

图 7.3

图 7.4

7.3　runC

经过 7.2 节，实现了一个简单版本的容器，基本包含了一些简单的容器操作。通过逐步的实战学习，相信读者对于容器的实现原理及涉及的技术点已经有了一些深刻的认识。那么，这一节就来介绍一下目前业界主流使用的容器运行引擎 runC。

7.3.1　简介

在过去的五年里，Linux 已经逐步增加了 Cgroups、Namespace、 Seccomp、capability、Apparmor 等一些功能，这些特性使得容器技术得以在 Linux 上快速发展。Docker 重度使用这些特性，而且目前风靡大江南北。实际上，容器技术是一系列晦涩难懂甚至有些神秘的系统特性的集合，因此 Docker 公司将这些底层的技术合并在一起，开源出了一个项目 runC。

实际上，runC 是由 Docker 公司 libcontainer 项目发展而来的，目前托管于 OCI 组织。Linux 基金会在 2015 年 6 月成立了 OCI（Open Container Initiative）组织，旨在围绕容器格式定义和运行时的配置制定一个开放的工业化标准。该组织主要由 Docker、Google、IBM、Microsoft、Red Hat 和其他许多合作伙伴创立。

runC 是一个轻量级的容器运行引擎，包括所有 Docker 使用的和容器相关的系统调用的代码，其基本功能点如下。

- 完全支持 Linux Namespace，包括 User Namespace。
- 原生支持所有 Linux 提供的安全特性：Selinux、Apparmor、Seccomp、control groups、capability、pivot_root 等。只要是 Linux 能做的，runC 都能做。
- 在 CRIU 项目的支持下原生支持容器热迁移。
- 一份正式的容器标准，由 Open Container Project 管理挂靠在 Linux 基金会下，可以说这是真正的业界标准。

可以这样理解，runC 的目标就是去构造到处都可以运行的标准容器。

7.3.2　OCI 标准包（bundle）

一个标准的容器运行时需要文件系统，也就是镜像。那么，OCI 是怎么定义一个基本的容器运行包的呢？这个容器标准包的定义仅仅考虑如何把容器和它的配置数据存储到磁盘上以便运行时读取。一般来说，应该包括如下 2 个模块。

- config.json 包括容器的配置数据。这个文件必须在容器的 root 文件系统内。

- 一个文件夹，代表容器的 root 文件系统。这个文件夹的名字理论上是可以随意的，但是按照一般命名规则，叫 rootfs 比较合适。当然，这个文件夹内必须包含上面提到的 config.json。

7.3.3 config.json

config.json 包含容器必需的元信息，主要包括容器需要去运行的进程、环境变量、沙盒环境等。下面把 config.json 含有的一些元素讲解一下。

- ociVersion：这里是指定 OCI 容器的版本号。
- root：配置容器的 root 文件系统。
  - path 指定 root 文件系统的路径，可以是以 / 开头的绝对路径，也可以是相对路径。
  - readonly 如果为 true，那么 root 文件系统在容器内就是只读的，默认是 false。

举例如下。

```
"root": {
    "path": "rootfs",
    "readonly": true
}
```

7.3.4 mounts

mounts 配置额外的挂载点。

- destination：挂载点在容器内的目标位置，必须是绝对路径。
- type：需要挂载的文件系统类型。其中类型必须是 Linux Kernel 支持的类型，比如 minix、ext2、ext3、jfs、xfs、reiserfs、proc、nfs。
- source：设备名或文件名。
- options：挂载点需要的额外信息。

举例如下。

```
"mounts": [
    {
        "destination": "/tmp",
        "type": "tmpfs",
        "source": "tmpfs",
        "options": ["nosuid","strictatime","mode=755","size=65536k"]
    },
    {
```

```
        "destination": "/data",
        "type": "bind",
        "source": "/volumes/testing",
        "options": ["rbind","rw"]
    }
]
```

7.3.5 process

process 配置容器进程信息如下。

- terminal：指定是否需要连接一个终端到此进程，默认是 false。
- consoleSize：在 terminal 连接时，用来指定控制台的大小。它包含下面两个属性。
  - height
  - width
- cwd：可执行文件的工作目录，这个路径必须是绝对路径。
- env：包含一系列需要传递给进程的环境变量，其中变量格式必须是 KEY=value 格式。
- args：传递给可执行文件的参数。
- capabilities：是一系列指定给容器进程的 capabilities 值。
- rlimits：限制容器内执行的进程资源使用量。

7.3.6 user

指定容器内运行进程的用户信息。

- uid 指定用户 ID。
- gid 指定 group ID。
- additionalGids 指定附加的 groups ID。

举例如下。

```
"process": {
    "terminal": true,
    "consoleSize": {
        "height": 25,
        "width": 80
    },
    "user": {
        "uid": 1,
        "gid": 1,
```

```
        "additionalGids": [5, 6]
    },
    "env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "TERM=xterm"
    ],
    "cwd": "/root",
    "args": [
        "sh"
    ],
    "apparmorProfile": "acme_secure_profile",
    "selinuxLabel": "system_u:system_r:svirt_lxc_net_t:s0:c124,c675",
    "noNewPrivileges": true,
    "capabilities": [
        "CAP_AUDIT_WRITE",
        "CAP_KILL",
        "CAP_NET_BIND_SERVICE"
    ],
    "rlimits": [
        {
            "type": "RLIMIT_NOFILE",
            "hard": 1024,
            "soft": 1024
        }
    ]
}
```

7.3.7 hostname

hostname 配置容器的主机名，只有容器创建了 UTS Namespace 才可以指定。

7.3.8 platform

platform 指定容器运行的系统信息，其中的两个参数如下。

- os 指定容器运行的系统类型。
- arch 指定系统的架构。

举例如下。

```
"platform": {
    "os": "linux",
    "arch": "amd64"
}
```

7.3.9　钩子（Hook）

配置文件内还提供了钩子的特性，它可以让开发者扩展容器运行态的动作，可以在容器运行前和停止后执行一些命令，这样用户可以做一些复杂的网络配置和 volume 的垃圾收集等动作。

- prestart: pre-start 钩子是在容器进程创建后执行的，但是在用户还没有开始执行前触发的。在 Linux 上，它是在 Namespace 创建成功后触发的，因此它能提供一个配置容器初始化环境的机会。
- poststart：post-start 钩子是在用户进程启动之后执行的。这个钩子可以用来告诉用户进程已经启动起来了。
- poststop：post-stop 钩子是在容器进程停止后执行的。这个钩子可以用来清理容器运行中产生的垃圾。

举例如下。

```
"hooks": {
        "prestart": [
            {
                "path": "/usr/bin/fix-mounts",
                "args": ["fix-mounts", "arg1", "arg2"],
                "env":  [ "key1=value1"]
            },
            {
                "path": "/usr/bin/setup-network"
            }
        ],
        "poststart": [
            {
                "path": "/usr/bin/notify-start",
                "timeout": 5
            }
        ],
        "poststop": [
            {
                "path": "/usr/sbin/cleanup.sh",
                "args": ["cleanup.sh", "-f"]
            }
        ]
    }
```

path 是需要执行脚本的路径。args 和 env 都是可选参数，timeout 是执行脚本的超时时间。这样就将 runC 基本的运行态所需要的配置和配置的信息讲解完了，下面会通过一个容器

的创建过程来讲解 runC 的源码。

7.4 runC 创建容器流程

前面介绍了 runC 的一些基本结构，那么 runC 到底是怎么根据配置文件创建容器的呢？本节将从源码分析的角度大概讲解一下 runC 创建容器的流程。

输入 runc run <container-id> 就会根据当前路径下面的 config.json 文件去创建一个容器。这里主要来介绍一下 runC 里面的 createContainer 流程，首先来看一下函数定义。

```
func createContainer(context *cli.Context, id string, spec *specs.Spec)
(libcontainer.Container, error) {
    config, err := specconv.CreateLibcontainerConfig(&specconv.CreateOpts{
        CgroupName:       id,
        UseSystemdCgroup: context.GlobalBool("systemd-cgroup"),
        NoPivotRoot:      context.Bool("no-pivot"),
        NoNewKeyring:     context.Bool("no-new-keyring"),
        Spec:             spec,
    })
    if err != nil {
        return nil, err
    }

    factory, err := loadFactory(context)
    if err != nil {
        return nil, err
    }
    return factory.Create(id, config)
}
```

createContainer 函数的参数列表接收上下文和关于容器的描述 spec，然后根据 spec 描述来配置容器需要的信息，最后把这些配置信息传递给 factory 的 create 方法。factory 可以基于很多系统实现，比如 Linux、Solaris、Windows、UNIX，这里主要看一下基于 Linux 的实现。

```
func (l *LinuxFactory) Create(id string, config *configs.Config) (Container,
error) {
    //检查配置信息
    if err := l.Validator.Validate(config); err != nil {
        return nil, newGenericError(err, ConfigInvalid)
    }
    logrus.Infof("Factory create containerRoot %s", containerRoot)
    //创建容器 root filesystem
    if err := os.MkdirAll(containerRoot, 0711); err != nil {
        return nil, newGenericError(err, SystemError)
```

```
    }
    if err := os.Chown(containerRoot, uid, gid); err != nil {
        return nil, newGenericError(err, SystemError)
    }
    fifoName := filepath.Join(containerRoot, execFifoFilename)
    logrus.Infof("fifoName %s", fifoName)
    oldMask := syscall.Umask(0000)
    // 创建进程间通信管道
    if err := syscall.Mkfifo(fifoName, 0622); err != nil {
        syscall.Umask(oldMask)
        return nil, newGenericError(err, SystemError)
    }
    syscall.Umask(oldMask)
    if err := os.Chown(fifoName, uid, gid); err != nil {
        return nil, newGenericError(err, SystemError)
    }
    // 生成包含容器信息的 struct
    c := &linuxContainer{
        id:            id,
        root:          containerRoot,
        config:        config,
        initArgs:      l.InitArgs,
        criuPath:      l.CriuPath,
        cgroupManager: l.NewCgroupsManager(config.Cgroups, nil),
    }
    return c, nil
}
```

这里截取了 Create 函数实现的一部分，其实主要工作就是检查容器配置，然后根据目录结构初始化一下容器的 root file system，最后把包含所有信息的 struct 返回。

容器信息创建完毕，就需要真正创建容器进程了，下面列出创建容器进程的 newParentProcess 函数。

```
func (c *linuxContainer) newParentProcess(p *Process, doInit bool) (parentProcess,
error) {
    // 创建匿名管道用于父子进程通信
    parentPipe, childPipe, err := newPipe()
    rootDir, err := os.Open(c.root)
    // 创建 command 信息
    cmd, err := c.commandTemplate(p, childPipe, rootDir)
    // 返回创建好的初始化进程信息
    return c.newInitProcess(p, cmd, parentPipe, childPipe, rootDir)
}
```

可以看到，newParentProcess 函数里面最重要的就是创建容器所属的 command 信息，下面来仔细看一下它的实现。

```
func (c *linuxContainer) commandTemplate(p *Process, childPipe, rootDir *os.File)
(*exec.Cmd, error) {
    // 创建 command
    cmd := exec.Command(c.initArgs[0], c.initArgs[1:]...)
      logrus.Infof("command template args1 %s   args2   %v", c.initArgs[0],
c.initArgs[1:])
    cmd.Stdin = p.Stdin
    cmd.Stdout = p.Stdout
    cmd.Stderr = p.Stderr
    cmd.Dir = c.config.Rootfs
    if cmd.SysProcAttr == nil {
        cmd.SysProcAttr = &syscall.SysProcAttr{}
    }
    cmd.ExtraFiles = append(p.ExtraFiles, childPipe, rootDir)
    cmd.Env = append(cmd.Env,
        fmt.Sprintf("_LIBCONTAINER_INITPIPE=%d", stdioFdCount+len(cmd.
ExtraFiles)-2),
        fmt.Sprintf("_LIBCONTAINER_STATEDIR=%d", stdioFdCount+len(cmd.
ExtraFiles)-1))
    // NOTE: when running a container with no PID namespace and the parent
    // process spawning the container is
    // PID1 the pdeathsig is being delivered to the container's init process by
    // the kernel for some reason
    // even with the parent still running.
    if c.config.ParentDeathSignal > 0 {
        cmd.SysProcAttr.Pdeathsig = syscall.Signal(c.config.ParentDeathSignal)
    }
    return cmd, nil
}
```

这段代码就不多做解释了，看过本书前面的章节应该就能明白，这和前面创建容器初始化进程是相似的流程，只是多加了一些环境变量和参数。容器创建其实就参考了 runC 实现。

最后，来看一下最终的 start 是如何实现的。

```
func (c *linuxContainer) start(process *Process, isInit bool) error {
    // 创建初始化进程
    parent, err := c.newParentProcess(process, isInit)
    logrus.Infof("libcontainer start %++v", parent)
    // 下面的 start 真正开启了容器进程的启动
    if err := parent.start(); err != nil {
        // terminate the process to ensure that it properly is reaped.
```

```
        if err := parent.terminate(); err != nil {
            logrus.Warn(err)
        }
        return newSystemErrorWithCause(err, "starting container process")
    }
```

至此，就完成了容器的初始化进程启动。下面会再次调用 runC 的 init 方法完成容器初始化进程的启动。这个参数在 factory_linux.go 里面有体现。

```
// New returns a linux based container factory based in the root directory and
// configures the factory with the provided option funcs.
func New(root string, options ...func(*LinuxFactory) error) (Factory, error) {
    if root != "" {
        if err := os.MkdirAll(root, 0700); err != nil {
            return nil, newGenericError(err, SystemError)
        }
    }
    l := &LinuxFactory{
        Root:      root,
        InitArgs:  []string{"/proc/self/exe", "init"},
        Validator: validate.New(),
        CriuPath:  "criu",
    }
    return l, nil
}
```

在 New 函数中，可以看到熟悉的 /proc/self/exe，后面的参数是 init，其实架构和 mydocker 一样，也会重新运行 runC 的 init 方法来初始化容器的进程。

代码读到这里，应该可以大概理解 runC 创建容器的整个过程了，如下。

1. 读取配置文件。
2. 设置 rootFileSystem。
3. 使用 factory 创建容器，各个系统平台均有不同实现。
4. 创建容器的初始化进程 process。
5. 设置容器的输出管道，主要是 Go 的 pipes。
6. 执行 Container.Start() 启动物理的容器。
7. 回调 init 方法重新初始化容器进程。
8. runC 父进程等待子进程初始化成功后退出。

可以看到，具体的执行流程涉及 3 个概念：process、container、factory。 factory 用来创建容器，process 负责进程之间的通信和启动容器。

7.5 Docker containerd 项目介绍

美国时间 2016 年 12 月 14 日，Docker 公司宣布将 containerd 从 Docker 中分离，并捐赠到一个新的开源社区独立发展和运营。containerd 可以作为 daemon 程序运行在 Linux 和 Windows 上，管理机器上所有容器的生命周期。阿里云、AWS、Google、IBM 和 Microsoft 作为初始成员，会为项目提供功能开发和维护。

containerd 对于很多人来说还是很陌生。Docker 公司为什么会大张旗鼓地宣布这个开源项目，并能得到业界巨大的反响呢？

实际上，早在 2016 年 3 月，Docker 1.11 的 Docker 里就包含了 containerd，而现在则是把 containerd 从 Docker 里彻底剥离出来，作为一个独立的开源项目独立发展，目标是提供一个更加开放、稳定的容器运行基础设施。和原先包含在 Docker 里的 containerd 相比，独立的 containerd 将具有更多的功能，可以涵盖整个容器运行时管理的所有需求。

containerd 并不是直接面向最终用户的，而是主要用于集成到更上层的系统里，比如 Swarm、Kubernetes、Mesos 等容器编排系统。containerd 以 daemon 的形式运行在系统上，通过 unix domain socket 暴露底层的 gRPC API，上层系统可以通过这些 API 管理机器上的容器。每个 containerd 只负责一台机器，Pull 镜像、对容器的操作（启动、停止等）、网络、存储都是由 containerd 完成的。具体运行容器由 runC 负责，实际上只要是符合 OCI 规范的容器都可以支持。containerd 项目架构图如图 7.5 所示。

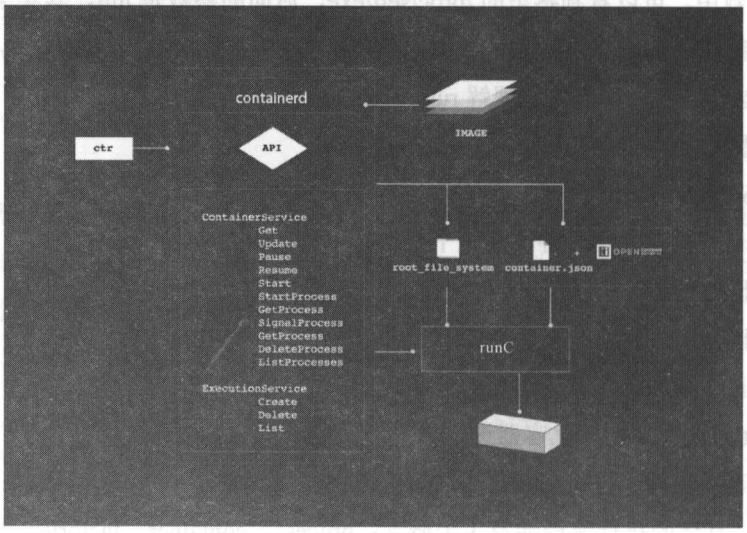


图 7.5

这对于社区和整个 Docker 生态来说是一件好事。对于 Docker 社区的开发者来说，独立的 containerd 更简单清晰，基于 containerd 增加新特性也会比以前容易。

对于容器编排服务来说，运行时只需要使用 containerd + runC，则会更加轻量、容易管理。而独立之后，containerd 的特性演进可以和 Docker 分开，专注容器运行时的管理，这样可以更稳定。在向后兼容上也可以做得更好，containerd 第一个正式版本 1.0 Release 之后将提供一年的支持，包括安全更新和 Bugfix，而每次升级也会向后兼容一个小版本。

Docker 为了表示对于社区和生态的诚意，特意强调了 containerd 中立的地位，符合各方利益。可以预见，containerd 将成为 Docker 平台的一个重要组件。阿里云、AWS、Google、IBM 和 Microsoft 将参与到 containerd 的开发中。

为了让大家更好地理解 containerd 的功能和架构，下面从更细节的角度看一下 containerd。

7.5.1　架构

图 7.6 是 containerd 的架构图。中间一层里包含了 3 个子系统，从这里可以看出 containerd 支持的能力有如下 3 个。

- Distribution：和 Docker Registry 打交道，拉取镜像。
- Bundle：管理本地磁盘上镜像的子系统。
- Runtime：创建容器、管理容器的子系统。

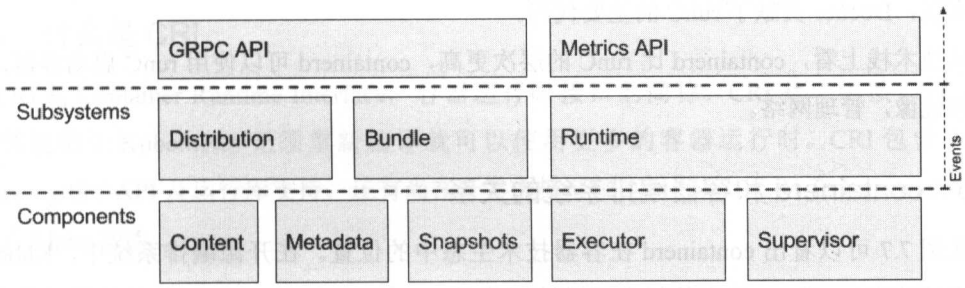


图 7.6

可以看出 containerd 非常干净，提供的都是运行时真正需要的功能。

7.5.2　特性和路线图

该项目的特性和路线图如下。

- 支持 OCI 镜像。

- 支持 OCI 运行时（runC）。
- 支持镜像的 pull/push 操作。
- 容器运行时和生命周期管理。
- 网络原语：创建 / 修改 / 删除接口。
- 让容器加入已有的 Network Namespace。
- 使用“内容可寻址”存储支持全局镜像多租户共享。

containerd 的当前版本是 0.2.4，是从 Docker 中剥离出来的一个功能子集。当功能覆盖前面的特性列表时，containerd 版本更新到 1.0，之后会保证 API 的稳定性，并提供 1 年的 LTS。

7.5.3 containerd 和 Docker 之间的关系

Docker 包含 containerd，containerd 专注于运行时的容器管理，而 Docker 除了容器管理之外，还可以完成镜像构建之类的功能。

containerd 提供的 API 偏底层，不是给普通用户直接用的。对于普通用户来说，可以继续使用 Docker。容器编排系统的开发者才需要 containerd，比如阿里云容器服务团队。

7.5.4 containerd、OCI 和 runC 之间的关系

OCI 是一个标准化的容器规范，包括运行时规范和镜像规范。runC 是基于此规范的一个参考实现，Docker 贡献了 runC 的主要代码。

从技术栈上看，containerd 比 runC 的层次更高，containerd 可以使用 runC 启动容器，还可以下载镜像，管理网络。

7.5.5 containerd 和容器编排系统的关系

从图 7.7 可以看出 containerd 在容器技术生态中的位置。在开源编排系统中，Kubernetes 现在直接使用 Docker，将来的版本可以转而使用 containerd；Mesos 和其他的编排引擎也可以使用 containerd 而不是直接使用 Docker。

对于云计算开发商来说，也可以非常方便地基于 containerd 提供定制化的容器网络、容器存储和编排方案。

containerd 在容器生态中扮演的角色

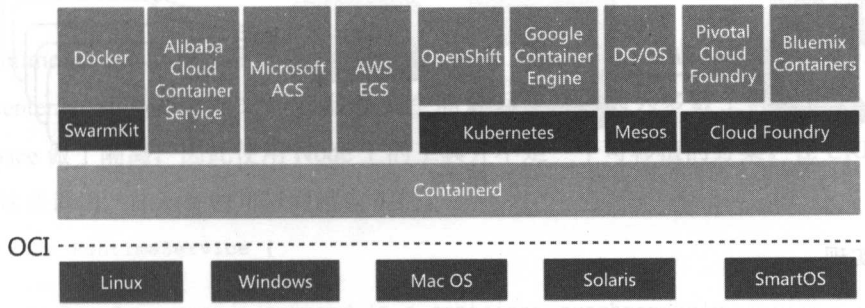


图 7.7

containerd 的目标是提供一个更加开放、稳定的容器运行基础设施。一方面，客户最终将受益于一个稳定且有良好支持的容器基础设施。另一方面，各家厂商可以利用 containerd 作为一个标准化、灵活的容器操作层，可以非常方便地提供定制化的网络、存储和容器编排。这对构建一个开放和健康的容器生态具有重要意义。

7.6　Kubernetes CRI 容器引擎

7.6.1　什么是 CRI

CRI 是 Container Runtime Interface，容器运行时接口的简称。CRI 是一组接口规范，这一插件规范让 Kubernetes 无须重新编译就可以使用更多的容器运行时。CRI 包含 Protocol Buffers、gRPC API 及运行库支持，还有尚在开发的标准规范和工具。CRI 在 Kubernetes 1.5 中发布了 Alpha 版本。

CRI 概览

Kubelet 通过 gRPC 框架与 CRI shim 进行通信，CRI shim 通过 Unix Socket 启动一个 gRPC server 提供容器运行时服务，Kubelet 作为 gRPC client，通过 Unix Socket 与 CRI shim 通信。gRPC server 使用 protocol buffers 提供两类 gRPC service：ImageService 和 RuntimeService。ImageService 提供从镜像仓库拉取镜像、删除镜像、查询镜像信息的 RPC 调用功能。RuntimeService 提供容器的相关生命周期管理（容器创建、修改、销毁等）及容器的交互操作（exec/attach/port-forward），其结构如图 7.8 所示。

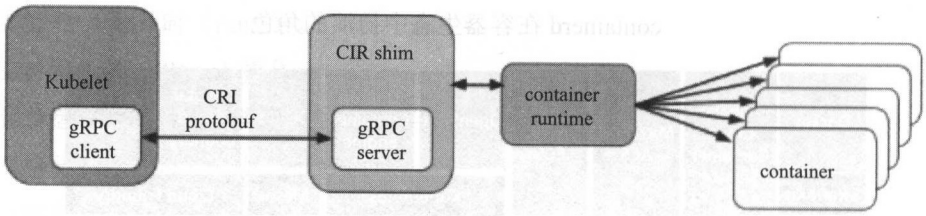


图 7.8

接口与实现

CRI 最核心的两个概念就是 PodSandbox 和 container。Pod 由一组应用容器组成，这些应用容器共享相同的环境与资源约束，这个共同的环境与资源约束被称为 PodSandbox。由于不同的容器运行时对 PodSandbox 的实现不一样，因此 CRI 留有一组接口给不同的容器运行时自主发挥，例如 Hypervisor 将 PodSandbox 实现成一个虚拟机，Docker 则将 PodSandbox 实现成一个 Linux 命名空间。

RuntimeService

Kubelet 在创建一个 Pod 前首先需要调用 RuntimeService。RunPodSandbox 为 Pod 创建一个 PodSandbox，这个过程包含初始化 Pod 网络、分配 IP、激活沙箱等。然后，Kubelet 会调用 CreateContainer、StartContainer、StopContainer、RemoveContainer 对容器进行创建、启动、停止、删除等操作。当 Pod 被删除时，会调用 StopPodSandbox、RemovePodSandbox 来销毁 Pod。Kubelet 的职责在于 Pod 的生命周期的管理，包含健康监测与重启策略控制，并且实现容器生命周期管理中的各种钩子。

```
service RuntimeService {
    // Sandbox operations.
    rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandboxResponse) {}
    rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSandboxResponse) {}
    rpc RemovePodSandbox(RemovePodSandboxRequest) returns (RemovePodSandboxResponse) {}
    rpc PodSandboxStatus(PodSandboxStatusRequest) returns (PodSandboxStatusResponse) {}
    rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSandboxResponse) {}
    // Container operations.
    rpc CreateContainer(CreateContainerRequest) returns (CreateContainerResponse) {}
    rpc StartContainer(StartContainerRequest) returns (StartContainerResponse) {}
    rpc StopContainer(StopContainerRequest) returns (StopContainerResponse) {}
    rpc RemoveContainer(RemoveContainerRequest) returns (RemoveContainerResponse) {}
    rpc ListContainers(ListContainersRequest) returns (ListContainersResponse) {}
    rpc ContainerStatus(ContainerStatusRequest) returns (ContainerStatusResponse) {}
```

```
    ...
}
```

RuntimeService 还定义了与 Pod 中容器进行交互的接口。目前，Kubelet 使用容器本地方法或 nsenter/socat 两种方法来与容器命名空间进行交互。因为多数工具假设 Pod 利用 Linux Namespace 做了隔离，因此使用 Node 上的工具并不是一个可移植的方案。在 CRI 中，显式地定义了这些调用，让运行时可以做特定实现。

```
service RuntimeService {
    ...
    // ExecSync runs a command in a container synchronously.
    rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
    // Exec prepares a streaming endpoint to execute a command in the container.
    rpc Exec(ExecRequest) returns (ExecResponse) {}
    // Attach prepares a streaming endpoint to attach to a running container.
    rpc Attach(AttachRequest) returns (AttachResponse) {}
    // PortForward prepares a streaming endpoint to forward ports from a PodSandbox.
    rpc PortForward(PortForwardRequest) returns (PortForwardResponse) {}
    ...
}
```

ImageService

为了启动一个容器，CRI 还需要执行镜像相关的操作，比如镜像的拉取、查看、移除等操作，因此 CRI 也定义了一组 ImageService 接口。但是容器的运行不需要镜像构建操作，所以 CRI 接口并不包含 buildImage 相关操作，镜像的构建需要使用外部工具如 Docker 来完成。

```
service ImageService{
    // ListImages lists existing images.
    ListImages(context.Context, *ListImagesRequest) (*ListImagesResponse, error)
    // ImageStatus returns the status of the image. If the image is not
    // present, returns a response with ImageStatusResponse.Image set to
    // nil.
     ImageStatus(context.Context, *ImageStatusRequest) (*ImageStatusResponse,
error)
    // PullImage pulls an image with authentication config.
    PullImage(context.Context, *PullImageRequest) (*PullImageResponse, error)
    // RemoveImage removes the image.
    // This call is idempotent, and must not return an error if the image has
    // already been removed.
     RemoveImage(context.Context, *RemoveImageRequest) (*RemoveImageResponse,
error)

}
```

LogService

LogService 定义了容器的 stdout/stderr 应该如何被 CRI 处理的相关规范。非 stdout/stderr 的日志处理不在 CRI 处理范围之内。CRI 的日志处理需要解决如下 3 个问题。

- 为 CRI 兼容的运行时提供相关方法来支持现存的日志特性，如支持 kubectl logs 及 docker log drivers 等。
- 允许 Kubelet 管理容器日志的生命周期，从而实现一个更优的磁盘管理策略。这就需要容器的生命周期和容器的日志之间进行解耦。
- 不论用户使用什么样的容器运行时，允许日志收集器更容易地与 Kubelet 集成，同时保证存取的高效性。

CRI 通过如下 2 个方面来满足这些需求。

- 强制指定日志应该存放在本地文件系统的某个位置，并且 Kubelet 和日志收集器能方便地直接访问该日志文件，如 /var/log/pods/<podUID>/<containerName>_<instance#>.log，CRI 端通过在创建 PodSandbox 的时候将这个目录指定给容器运行时实现，日志收集器也可以方便地通过 Pod 的元数据获取该信息。
- 运行时需要按照 Kubelet 能够理解的方式输出日志。运行时需要为每条日志添加 RFC 3339 Nano 时间戳及 stream 类型，并以一个新行结束。

```
2016-10-06T00:17:10.113242941Z stderr The content of the log entry 1
2016-10-06T00:17:09.669794202Z stdout The content of the log entry 2
```

短期来看，docker-CRI 实现只是部分支持以上几个原则，采取的方式一个是为 Kubelet 创建日志文件的符号链接，另一个是在 Kubelet 中添加 json 格式的日志文件处理支持。长期来看，CRI 可能会支持一个自定义的日志处理插件或启动一个额外的进程来复制或装饰原始日志。

7.6.2 为什么需要 CRI

在 CRI 存在之前，容器运行时（Docker/rkt）需要通过实现 Kubelet 里面 high-level 的接口才能集成进来。这样的集成使得成本很高，需要开发人员了解整个 Kubelet 的架构，并且会将代码合并到 Kubelet 主项目里面。更重要的是，这非常不利于 Kubelet 的扩展性，因为每个新的改动都会不断地增加维护难度。Kubernetes 的目标是具有高度的可扩展性，而 CRI 为方便使用一个可插拔的容器运行时（Docker/rkt）并构建一个健康的 Kubernetes 生态迈出了一小步。原因总结起来有如下 3 条。

- 不是所有的容器运行时都原生地支持 Pod 概念。为了支持所有这些 Pod 特性，当这些容器运行时与 Kubernetes 集成时，需要花费大量的工作时间来实现一个 Shim。
- 高层次的接口让代码共享和重用变得困难。
- Pod Spec 演化速度非常快。Pod 经常被添加新的特征，任何 Pod 级别的特性修改或增加都会导致容器运行时的修改。

7.6.3　为什么 CRI 是接口且是基于容器的而不是基于 Pod 的

Kubernetes 有 Pod 级别的资源接口，如果直接对 Pod 做抽象，CRI 就可以自己实现容器的控制逻辑和状态转换，这样就可以极大地简化 API，让 CRI 能适应更多的容器运行时，但是 Kubernetes 团队最终放弃了这个想法，原因有如下 2 点。

- 首先，Kubelet 有很多 Pod 级别的功能和机制（例如循环崩溃的处理），交给容器运行时实现的话，会造成很重的负担。
- 其次，更重要的是，Pod 标准还在高速前进。很多新功能（例如容器初始化）是由 Kubelet 直接管理容器的，而无须容器运行时进行变更。

CRI 选择了围绕容器进行实现，这样容器运行时能够共享这些通用特性，获得更好的开发进度。

7.6.4　如何使用 CRI

Kubernetes 1.5.0 目前还未正式启用 CRI，正处于 Working on 阶段，Kubernetes 会逐步支持 CRI 规范，第一步会从使用 CRI 方式集成 Docker 运行时开始，并且可以通过 Kubelet 参数来开启这项处于实验阶段的功能。

Docker CRI 实现

- 设置 Kubelet flag。

```
--experimental-cri=true --container-runtime=docker
```

Kubelet 会启动一个 Local Unix Socket 的 gRPC CRI server 进行 PodSandBox、Pod、container 的生命周期管理。

cri-o CRI 实现

- 在节点上启动 runtime service，如 cri-o，请查看 github.com/kubernetes-incubator/cri-o。
- 设置 Kubelet flag。

```
--experimental-cri=true --container-runtime=remote --container-runtime-
```

```
endpoint=unix:///var/run/oci.sock --image-service-endpoint=unix:///var/run/oci.sock
```

7.6.5 CRI 的目标

定义 CRI 的目的在于以下 3 点。

- 提升 Kubernetes 的可扩展性，让更多的容器运行时更容易集成到 Kubernetes 中。
- 提升 Pod Feature 的更新迭代效率。
- 构建易于维护的代码体系。

CRI 不会做以下 5 件事。

- 建议如何与新的容器运行时集成，例如决定 Container Shim 应该在哪里实现。
- 提供新接口的版本管理。
- 提供 Windows container 支持。本接口不会在 Windows container 支持方面花费太多，但是 CRI 会尽量做到更加易于扩展来让 Windows container 特性更容易被添加进来。
- 重新定义 Kubelet 的内部运行时相关接口。尽管会增加 Kubelet 的可维护性，但这不是 CRI 的工作。
- 提升 Kubelet 的效率和性能。

7.6.6 已知的问题

CRI 已知的问题

- CRI 规范还未定义容器统计信息的相关内容。
- 现存的日志处理工具（如 fluentd、GCL 等）还不支持新的日志规范。
- CRI 可能与其他的实验性质功能不兼容（如 Seccomp）。
- 服务处理端需要加固。如认证、避免在重定向 URL 中传递用户数据等。

Docker-CRI 实现已知问题

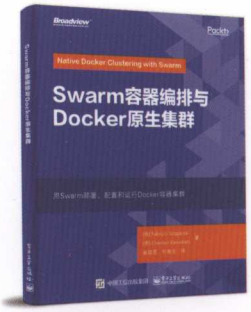
- 仅仅支持 Docker v1.11 和 v1.12。
- 网络方面不支持主机端口映射及带宽控制。
- 容器交互方面还不支持 nsenter 作为 exec 的 handler。

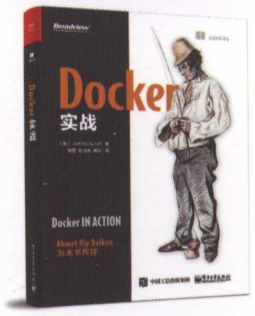
7.7　小结

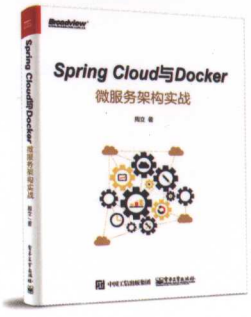
任何系统在演化的过程中，功能会不断地丰富，继之而来的是系统实现的臃肿，接口抽象与插件就成了实现级别的解决方案，这是演化的必经之路。CRI 目前还处于 Alpha 阶段，它为 Kubernetes 的可扩展性与易维护打下了坚实的基础，同时也面临许多难题需要逐个解决。相信 CRI 未来会提供一个更酷的运行时体验。

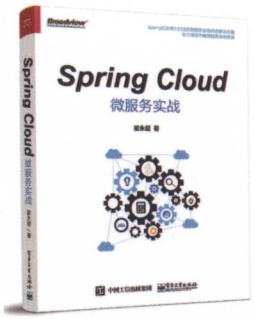
本章首先使用自己创建的 mydocker 做了两个 demo。第一个实现了运行一个 Nginx 镜像并且对外访问的功能，第二个实现了 flask + redis 的计数器，一个 flask 容器连接一个 redis 容器，并且联合对外服务。到此为止，我们的容器之路算是告一段落了。之后，又介绍了 Docker 使用的容器运行时引擎 runC，以及从源码级解释了 runC 创建一个容器的流程。最后分别介绍了 Docker 最新的单机容器引擎 containerd 和 Kubernetes 的容器运行引擎 CRI。虽然各家在单机容器运行引擎上各自发展，但是容器的原理依然不变。

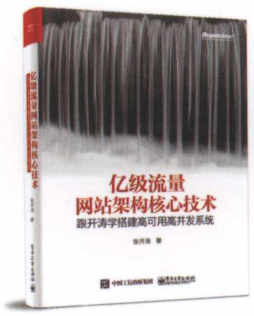
通过阅读本书，可以很好地了解到底什么是容器，一个容器是如何通过各种技术的组合运行起来并提供对外访问隔离的。这给读者在后面继续深入探索容器的核心内容打下了基础。本书的代码仅仅为了解释原理，并不具备生产使用条件，有兴趣的读者可以尝试继续添加功能并且强化它，非常欢迎大家继续开发贡献。


Broadview
Packt
Native Docker Clustering with Swarm
Swarm容器编排与
Docker原生集群


Broadview
Docker
实战
Docker IN ACTION


Broadview
Spring Cloud与Docker
微服务架构实战


Broadview
Spring Cloud
微服务实战


Broadview
亿级流量
网站架构核心技术
跟开涛学搭建高可用高并发系统

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

自己动手写Docker

容器技术发展日新月异，除了不断地跟进最新的版本和社区发展，最好的深入学习方式就是通过模仿来造一个类似的轮子。

本书结合了几位作者在容器领域和阿里云上的生产实践，涉及了从入门的容器技术和 Go 原理到最新的容器领域规范和开源项目 OCI、containerd、CRI 等，很精炼地将如何从零写一个 Docker 娓娓道来，非常有助于提高读者在 Docker 领域的深度动手能力。

——阿里巴巴高级技术专家，汤志敏

本书通过从头构建容器引擎、构造镜像，深入浅出地讲解了容器背后的原理，是一本不可多得的好书。

——阿里巴巴高级技术专家，戒空

随着 Docker 技术的不断发展，Docker 公司、阿里云及其他的云产品公司都推出了越来越成熟的、基于 Docker 的解决方案，一场 Docker 容器带来的技术变革正在兴起。本书内容由浅至深，通俗易懂，引导读者通过学习容器技术的实现细节，一步步去构建一个简单的容器，能帮助有一定 Docker 基础的工程师学习到更有实践性的经验，对刚接触 Docker 技术的工程师也很有参考价值。

——阿里巴巴技术专家，罗晶

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：徐津平
封面设计：吴海燕

上架建议：容器

ISBN 978-7-121-31786-6

定价：65.00元